C Language Reference

Microsoft® C/C++

Microsoft® C/C++

Version 7.0

C Language Reference

For MS-DOS® and Windows™ Operating Systems

Microsoft Corporation

Document No. LN24771-1191

10 9 8 7 6 5 4 3 2 1

Contents Overview

Contents

Tables

Tables

Introduction

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 for programmers to read and maintain.

C is a flexible language that leaves many programming decisions up to you. In keeping with this philosophy, 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. This manual provides information on the C language components and the features of the Microsoft implementation. The syntax for the C language is from the ANSI standard, although it is not part of the ANSI standard. Appendix A provides the syntax and a description of how to read and use the syntax definitions.

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. To perform such tasks, C programmers rely on run-time libraries. 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. See the *Microsoft C Run-Time Library Reference* or the online Help for information about the library routines provided with Microsoft C version 7.0.

This manual does not discuss programming with C++. See the *C++ Tutorial* and the *C++ Language Reference* for information about the C++ language.

Other tools required for programming with the Microsoft C compiler, such as LINK, LIB, CodeView, and the integrated programming environment called Programmer's WorkBench (PWB) are discussed in *Environment and Tools.*

To find information on a particular topic in the documentation, see the *Comprehensive Index and Errors Reference.*

ANSI Conformance

Microsoft C version 7.0 conforms to the standard for the C language as set forth in the American National Standard (hereinafter referred to as the ANSI C standard). Microsoft extensions to the ANSI C standard are noted in the text and syntax of this manual as well as in the online reference. Because the extensions are not a part of the ANSI C standard, their use may restrict portability of programs between systems. By default, the Microsoft extensions are enabled. To disable the extensions, specify /Za on the command line. With /Za, all non-ANSI code generates errors or warnings.

Scope and Organization of This Manual

The *C Language Reference* defines the C language as implemented by Microsoft Corporation. It is intended as a reference for programmers experienced in C or other programming languages. Knowledge of programming fundamentals is assumed. This manual is organized as follows:

Chapter 1, "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 2, "Program Structure," discusses the components and structure of C programs and explains how C source files are organized.

Chapter 3, "Declarations and Types," 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. Function prototypes are discussed in this chapter, as well as in Chapter 6, "Functions."

Chapter 4, "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 5, "Statements," describes C statements, which control the flow of program execution.

Chapter 6, "Functions," discusses C functions. In particular, this chapter explains function prototypes, parameters, and return values, as well as how to define, declare, and call functions.

Chapter 7, "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 Microsoft-specific instructions to the compiler that you may place in source files.

Appendix A, "C Language Syntax Summary," is the complete listing of the ANSI syntax as discussed in the rest of the manual. The Microsoft-specific features are noted. This appendix also begins with a description of how to read and use the ANSI C syntax.

NOTE The ANSI syntax in Appendix A is provided for information only. It is not part of the ANSI standard.

Appendix B, "Implementation-Defined Behavior," describes Microsoft's implementation of the areas that ANSI leaves open to each particular implementation. These items are listed in Appendix F, "Portability Issues," Section F.3, of the ANSI standard.

Appendix C, "Differences Between Microsoft C Versions 6.0 and 7.0," lists the new and changed features of the C compiler since Microsoft C version 6.0. This appendix does not include information about features of the C++ compiler.

Document Conventions

This book uses the following typographic conventions:

Example	Description
STDIO.H	Uppercase letters indicate filenames, segment names, registers, and terms used at the operating-system command level.
char, **_setcolor**, **__far**	Bold type indicates keywords, operators, language-specific characters, and library routines. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown. Many functions and constants begin with either a single or double underscore. These are part of the name and are mandatory. For example, to have the **__cplusplus** manifest constant be recognized by the compiler, you must enter the leading double underscore.
expression	Words in italics indicate placeholders for information you must supply, such as a filename.
[[*option*]]	Items inside double square brackets are optional.

Example	Description
#pragma pack {**1** | **2**}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless double square brackets ([[ ]]) surround the braces.
`#include <io.h>`	This font is used for examples, user input, program output, and error messages in text.
CL [[*option...*]] *file...*	Three dots (an ellipsis) following an item indicate that more items having the same form may appear.
`while() { . . . }`	A column or row of three dots tells you that part of an example program has been intentionally omitted.
CTRL+ENTER	Small capital letters are used to indicate the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second.
	The carriage-return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.
"argument"	Quotation marks enclose a new term the first time it is defined in text.
`"C string"`	Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form `" "` and `' '` rather than " " and ' '.
Color Graphics Adapter (CGA)	The first time an acronym is used, it is usually spelled out.
♦	This symbol denotes the end of a section of "Microsoft Specific" or "32-Bit Specific" information.

Elements of C

Chapter 1

This chapter describes the elements of the C programming language, including the names, numbers, and characters used to construct a C program. The ANSI C syntax labels these components "tokens." This chapter explains how to define these tokens and how the compiler evaluates them. The following topics are discussed in this chapter:

- Tokens
- Comments
- Keywords
- Identifiers
- Constants
- String literals
- Operators
- Punctuation

This chapter also includes reference tables for the C and Microsoft-specific keywords, trigraphs, escape sequences, and operator precedence, as well as the range for floating-point and integer constants.

1.1 Tokens

In a C source program, the basic element recognized by the compiler is the "token." A token is source-program text that the compiler does not break down into component elements.

Syntax

token :
  keyword
  identifier
  constant
  string-literal
  operator
  punctuator

Note See the introduction to Appendix B for an explanation of the ANSI grammar conventions.

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

White-Space Characters

Space, tab, linefeed, carriage-return, formfeed, vertical-tab, and newline characters are called "white-space characters" because they serve the same purpose as the spaces between words and lines on a printed page—they make reading easier. Tokens are delimited (bounded) by white-space characters and by other tokens, such as operators and punctuation. When parsing code, the C compiler ignores white-space characters unless you use them as separators or as components of character constants or string literals. Use white-space characters to make a program more readable. Note that the compiler also treats comments as white space.

Comments

A "comment" is a sequence of characters beginning with a forward slash/asterisk combination (/*) that is treated as a single white-space character by the compiler and is otherwise ignored. A comment can include any combination of characters from the representable character set, including newline characters, but excluding the "end comment" delimiter (*/). Comments can occupy more than one line but 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. The compiler ignores the characters in the comment.

Use comments to document your code. This example is a comment accepted by the compiler:

```
/* Comments can contain keywords such as
   for and while without generating errors. */
```

Comments can also appear on the same line as a code statement:

```
printf( "Hello\n" );  /* Comments can go here */
```

You may choose to precede functions with a descriptive comment block:

```
/* MATHERR.C illustrates writing an error routine
 * for math functions.
 * The error function must be:
 *  matherr
 *
 * To use matherr, you must turn on the No Extended Dictionary in
 * Library flag within the PWB environment (LINK Options from the
 * Options menu)or use the /NOE linker option outside the environment.
 * For example:
 *  CL matherr.c /link /NOE
 */
```

Since comments cannot contain nested comments, this example causes an error:

```
/* Comment out this routine for testing

    /* Open file */
    fh = _open( "myfile.c", _O_RDONLY );
    .
    .
    .
 */
```

The error occurs because the compiler recognizes the first `*/`, after the words `Open file`, as the end of the comment. It tries to process the remaining text and produces an error when it finds the `*/` outside a comment.

While you can use comments to render certain lines of code inactive for test purposes, the preprocessor directives **#if** and **#endif** and conditional compilation are a useful alternative for this task. For more information, see "Conditional Compilation" on page 202.

Microsoft Specific

The Microsoft compiler also supports single-line comments preceded by two forward slashes (*//*). If you compile with /Za (ANSI standard), these comments generate errors. These comments cannot extend to a second line.

```
// This is a valid comment in C 7.0
```

Comments beginning with two forward slashes (*//*) are terminated by the next newline character that is not preceded by an escape character. In the next example, the newline character is preceded by a backslash (\), creating an "escape sequence." This escape sequence causes the compiler to treat the next line as part of the previous line. (For information on escape sequences, see page 18.)

```
// my comment \
    i++;
```

Therefore, the `i++;` statement is commented out.

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za command-line option to disable these extensions.♦

Evaluation of Tokens

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. 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 ensure proper expression evaluation.

Microsoft Specific The C compiler treats a CTL+Z character as an end-of-file indicator. It ignores any text after CTRL+Z.♦

1.2 Keywords

"Keywords" are words that have special meaning to the C compiler. In translation phases 7 and 8, an identifier cannot have the same spelling and case as a C keyword. (For a description of translation phases, see page 190; for information on identifiers, see page 5.) The C language uses 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 7, "Preprocessor Directives and Pragmas").

Microsoft Specific The ANSI specification allows identifiers with two leading underscores to be reserved for compiler implementations. Therefore, the Microsoft convention is to precede Microsoft-specific keyword names with double underscores. For a description of the ANSI rules for naming identifiers, including the use of double underscores, see the next section.

The Microsoft C version 7.0 compiler also recognizes the following keywords. These words cannot be used as identifier names. Note that there is a double underscore preceding each keyword.

__asm	**__fastcall**	**__loadds**	**__self**
__based	**__finally**	**__near**	**__stdcall**
__cdecl	**__fortran**	**__pascal**	**__try**
__except	**__huge**	**__saveregs**	
__export	**__inline**	**__segment**	
__far	**__interrupt**	**__segname**	

Limited 32-bit support is available for **__based**. Seven of the above Microsoft-specific keywords are not supported in 32-bit compilations: **__far**, **__fortran**, **__huge**, **__near**, **__segment**, **__segname**, and **__self**. The **__except** and **__finally** keywords are not supported in 16-bit compilations.

For the Microsoft C compiler, extensions are enabled by default. You can make sure your program is fully portable by disabling the Microsoft extensions with the /Za command-line option. From within the Programmer's WorkBench, select Language Options and then C Compiler Options from the Options menu. Then select Additional Global Options from the C Compiler Options dialog box. Choose ANSI C in the Languages box; this disables the Microsoft extensions.♦

1.3 Identifiers

"Identifiers" or "symbols" are the names you supply for variables, types, functions, and labels in your program. Identifier names must differ in spelling and case from any keywords. You cannot use keywords (either C or Microsoft) as identifiers; they are reserved for special use. You create an identifier by specifying it in the declaration of a variable, type, or function. In this example, `result` is an identifier for an integer variable, and `main` and `printf` are identifier names for functions.

```
void main()
{
    int result;

    if ( result != 0 )
       printf( "Bad file handle\n" );
}
```

Once declared, you can use the identifier in later program statements to refer to the associated value.

A special kind of identifier, called a statement label, can be used in **goto** statements. (Declarations are described in Chapter 3. Statement labels are described in "The goto and Labeled Statements" on page 157.)

Syntax

identifier :
  nondigit
  identifier nondigit
  identifier digit

nondigit : one of
  _ 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

digit : one of
  0 1 2 3 4 5 6 7 8 9

The first character of an identifier name must be a *nondigit* (that is, the first character must be an underscore or an uppercase or lowercase letter). ANSI allows six significant characters in an external identifier's name and 31 for names of internal (within a function) identifiers. External identifiers (ones declared at global scope or declared with storage class **extern**) may be subject to additional naming restrictions because these identifiers have to be processed by other software such as linkers.

Microsoft Specific

The Microsoft C compiler allows 32 characters in an external identifier's name and 247 for internal identifier names. Compiling with the /W4 command-line option or selecting warning level 4 from the Compiler Options dialog box (which is accessed from selecting Language Options on the Option menu in PWB) generates warnings on any names that exceed the ANSI-specified length. If you are not concerned with ANSI compatibility, you can modify this default to a smaller number using the /H (restrict length of external names) option. ♦

The C compiler considers uppercase and lowercase letters to be distinct characters. This feature, called "case sensitivity," enables you to create distinct identifiers that have the same spelling but different cases for one or more of the letters.

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

```
add
ADD
Add
aDD
```

Microsoft Specific

Do not select names for identifiers that begin with two underscores or with an underscore followed by an uppercase letter. The ANSI specification allows identifier names that begin with these character combinations to be reserved for compiler use. Identifiers with file-level scope should also not be named with an underscore and a lowercase letter as the first two letters. Identifier names that begin with these characters are also reserved. By convention, Microsoft uses an underscore and an uppercase letter to begin macro names and double underscores for Microsoft-specific keyword names. To avoid any naming conflicts, always select identifier names that do not begin with one or two underscores, or names that begin with an underscore followed by an uppercase letter. ♦

The following are examples of valid identifiers that conform to either ANSI or Microsoft naming restrictions:

```
j
count
temp1
top_of_page
skip12
LastNum
```

Microsoft Specific

Although identifiers in source files are case sensitive by default, symbols in object files are not. The /Zc command-line option tells the compiler to ignore case for any identifier name declared with the **__pascal** keyword. The /Gc command-line option specifies the FORTRAN/Pascal calling convention, causing all function names to be translated to uppercase. (The **__pascal** and **__fortran** keywords perform the same operation on a function-by-function basis.) For information on the command-line options, see Chapter 13 in the *Environment and Tools* manual.

Externally linked identifiers may or may not be case sensitive, depending on whether you use the /NOIGNORECASE option when you invoke the linker. The default for the Microsoft linker is to ignore case, making externally linked identifiers case insensitive. Note that the external name of an identifier may be different from its internal name. This is only an issue when you link C object files with non-C object files. For more information about linking, see Chapter 14 in the *Environment and Tools* manual.

The "source character set" is the set of legal characters that can appear in source files. For Microsoft C, the source set is the standard ASCII character set. The source character set and execution character set include the ANSI ASCII characters used as escape sequences. See "Character Constants" on page 16 for information about the execution character set.♦

An identifier has "scope," which is the region of the program in which it is known, and "linkage," which determines whether the same name in another scope refers to the same identifier. These topics are explained in "Understanding Lifetime, Scope, Visibility, and Linkage" on page 34.

Multibyte and Wide Characters

A multibyte character is a character composed of sequences of one or more bytes. Each byte sequence represents a single character in the extended character set. Multibyte characters are used in character sets such as Kanji.

Wide characters are multilingual character codes that are always 16 bits wide. The type for character constants is **char**; for wide characters, the type is **wchar_t**. Since wide characters are always a fixed size, using wide characters simplifies programming with international character sets.

The wide-character-string literal `L"hello"` becomes an array of six integers of type **wchar_t**.

```
{L'h', L'e', L'l', L'l', L'o', 0}
```

The Unicode specification is the developing specification for wide characters. The run-time library routines for translating between multibyte and wide multibyte characters include **mblen, mbstowcs**, **mbtowc**, **wcstombs**, and **wctomb**.

Trigraphs

The source character set of C source programs is contained within the 7-bit ASCII character set but is a superset of the ISO 646-1983 Invariant Code Set. Trigraph sequences allow C programs to be written using only the ISO (International Standards Organization) Invariant Code Set. Trigraphs are sequences of three characters (introduced by two consecutive question marks) that the compiler replaces with their corresponding punctuation characters. You can use trigraphs in C source files with a character set that does not contain convenient graphic representations for some punctuation characters.

Table 1.1 shows the nine trigraph sequences. All occurrences in a source file of the punctuation characters in the first column are replaced with the corresponding character in the second column.

Table 1.1 Trigraph Sequences

Trigraph	Punctuation Character
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{

Table 1.1 Trigraph Sequences (*continued*)

Trigraph	**Punctuation Character**
??!	|
??>	}
??-	~

A trigraph is always treated as a single source character. The translation of trigraphs takes place in the first translation phase, before the recognition of escape characters in string literals and character constants. (See page 190 for information about translation phases.) Only the nine trigraphs shown in Table 1.1 are recognized. All other character sequences are left untranslated.

The character escape sequence, **\?**, prevents the misinterpretation of trigraph-like character sequences. (See page 18 for information about escape sequences.) For example, if you attempt to print the string `What??!` with this **printf** statement

```
printf( "What??!\n" );
```

the string printed is `What|` because `??!` is a trigraph sequence that is replaced with the `|` character. You need to write the statement as follows to correctly print the string:

```
printf( "What?\?!\n" );
```

In this **printf** statement, a backslash escape character in front of the second question mark prevents the misinterpretation of `??!` as a trigraph.

1.4 Constants

A "constant" is a number, character, or character string that can be used as a value in a program. Use constants to represent floating-point, integer, enumeration, or character values that cannot be modified.

Syntax

constant :
  floating-point-constant
  integer-constant
  enumeration-constant
  character-constant

Constants are characterized by having a value and a type. This section discusses floating-point, integer, and character constants. Enumeration constants are described in "Enumeration Declarations" on page 62.

Floating-Point Constants

A "floating-point constant" is a decimal number that represents a signed real number. The representation of a signed real number includes an integer portion, a fractional portion, and an exponent. Use floating-point constants to represent floating-point values that cannot be changed.

Syntax

floating-point-constant :
  fractional-constant exponent-part $_{\text{opt}}$ *floating-suffix* $_{\text{opt}}$
  digit-sequence exponent-part floating-suffix $_{\text{opt}}$

fractional-constant :
  digit-sequence $_{\text{opt}}$ **.** *digit-sequence*
  digit-sequence **.**

exponent-part :
  e *sign* $_{\text{opt}}$ *digit-sequence*
  E *sign* $_{\text{opt}}$ *digit-sequence*

sign : one of
  + –

digit-sequence :
  digit
  digit-sequence digit

floating-suffix : one of
  f l F L

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

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

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

Floating-point constants are positive unless they are preceded by a minus sign (–). In such case, the minus sign is treated as a unary arithmetic negation operator.

Floating-point constants have type **float**, **double**, or **long**. A floating-point constant without an **f**, **F**, **l**, or **L** suffix has type **double**. If the letter **f** or **F** is the suffix, the constant has type **float**. If suffixed by the letter **l** or **L**, it has type **long double**. For example:

```
100L  /* Has type long   */
100F  /* Has type float  */
100D  /* Has type double */
```

See "Storage of Basic Types" on page 98 for information about type **double**, **float**, and **long**.

You can omit the integer portion of the floating-point constant, as shown in the following examples. The number .75 can be expressed in many ways, including the following:

```
.0075e2
0.075e1
.075e1
75e-2
```

Microsoft Specific

Limits on the values of floating-point constants are given in the Table 1.2. The header file FLOAT.H that the setup program for Microsoft C/C++ version 7.0 installs in your \C700\INCLUDE directory contains this information.

Table 1.2 Limits on Floating-Point Constants

Constant	Meaning	Value
FLT_DIG	Number of digits, *q*, such that a floating-point number with *q* decimal digits can be rounded into a floating-point representation and back without loss of precision.	6
DBL_DIG		15
LDBL_DIG		18
FLT_EPSILON	Smallest positive number *x*, such that x1.0+x	1.192092896e-07F
DBL_EPSILON		2.2204460492503131e-016
LDBL_EPSILON		1.084202172485504434e-019L
FLT_MANT_DIG	Number of digits in the radix specified by **FLT_RADIX** in the floating-point significand. In Microsoft C++, the radix is 2; hence these values specify bits.	24
DBL_MANT_DIG		53
LDBL_MANT_DIG		64

Table 1.2 Limits on Floating-Point Constants *(continued)*

Constant	Meaning	Value
FLT_MAX **DBL_MAX** **LDBL_MAX**	Maximum representable floating-point number.	3.402823466e+38F 1.7976931348623158e+30 1.189731495357231765e+4932L
FLT_MAX_10_EXP **DBL_MAX_10_EXP** **LDBL_MAX_10_EXP**	Maximum integer such that 10 raised to that number is a representable floating-point number.	38 308 4932
FLT_MAX_EXP **DBL_MAX_EXP** **LDBL_MAX_EXP**	Maximum integer such that **FLT_RADIX** raised to that number is a representable floating-point number.	128 1024 16384
FLT_MIN **DBL_MIN** **LDBL_MIN**	Minimum positive value.	1.175494351e-38F 2.2250738585072014e-308 3.3621031431120935063e-4932L
FLT_MIN_10_EXP **DBL_MIN_10_EXP** **LDBL_MIN_10_EXP**	Minimum negative integer such that 10 raised to that number is a representable floating-point number.	-37 -307 -4931
FLT_MIN_EXP **DBL_MIN_EXP** **LDBL_MIN_EXP**	Minimum negative integer such that **FLT_RADIX** raised to that number is a representable floating-point number.	-125 -1021 -16381
FLT_RADIX **DBL_RADIX** **LDBL_RADIX**	Radix of exponent representation.	2 2 2
FLT_ROUNDS **DBL_ROUNDS** **LDBL_ROUNDS**	Rounding mode for floating-point addition.	1 (near) 1 (near) 1 (near)

Note that the information in Table 1.2 may differ in future implementations.♦

Integer Constants

An “integer constant” is a decimal (base 10), octal (base 8), or hexadecimal (base 16) number that represents an integral value. Use integer constants to represent integer values that cannot be changed.

Syntax

integer-constant :
  decimal-constant integer-suffix opt
  octal-constant integer-suffix opt
  hexadecimal-constant integer-suffix opt

decimal-constant :
  nonzero-digit
  decimal-constant digit

octal-constant :
  0
  octal-constant octal-digit

hexadecimal-constant :
  0x *hexadecimal-digit*
  0X *hexadecimal-digit*
  hexadecimal-constant hexadecimal-digit

nonzero-digit : one of
  1 2 3 4 5 6 7 8 9

octal-digit : one of
  0 1 2 3 4 5 6 7

hexadecimal-digit : one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-suffix :
  unsigned-suffix long-suffix opt
  long-suffix unsigned-suffix opt

unsigned-suffix : one of
  u U

long-suffix : one of
  l L

Integer constants are positive unless they are preceded by a minus sign (–). The minus sign is interpreted as the unary arithmetic negation operator. (See "Unary Operators" on page 122 for information about this operator.)

If an integer constant begins with the letters **0x** or **0X**, it is hexadecimal. If it begins with the digit **0**, it is octal. Otherwise, it is assumed to be decimal.

The following lines are equivalent:

```
0x1C   /* = Hexadecimal representation for decimal 28 */
034    /* = Octal representation for decimal 28 */
```

No white-space characters can separate the digits of an integer constant. These examples show valid decimal, octal, and hexadecimal constants.

```
/* Decimal Constants */
10
132
32179

/* Octal Constants */
012
0204
076663

/* Hexadecimal Constants */
0xa or 0xA
0x84
0x7dB3 or 0X7DB3
```

Integer Types

Every integer constant is given a type based on its value and the way it is expressed. You can force any integer constant to type **long** by appending the letter **l** or **L** to the end of the constant; you can force it to be type **unsigned** by appending **u** or **U** to the value. The lowercase letter **l** can be confused with the digit 1 and should be avoided. Some forms of **long** integer constants follow:

```
/* Long decimal constants */
10L
79L

/* Long octal constants */
012L
0115L

/* Long hexadecimal constants */
0xaL or 0xAL
0X4fL or 0x4FL
```

```
/* Unsigned long decimal constant */
776745UL
778866LU
```

The type you assign to a constant depends on the value the constant represents. A constant's value must be in the range of representable values for its type. A constant's type determines which conversions are performed when the constant is used in an expression or when the minus sign (–) is applied. This list summarizes the conversion rules for integer constants.

- The type for a decimal constant without a suffix is either **int**, **long int**, or **unsigned long int**. The first of these three types in which the constant's value can be represented is the type assigned to the constant.
- The type assigned to octal and hexadecimal constants without suffixes is **int**, **unsigned int**, **long int**, or **unsigned long int** depending on the size of the constant.
- The type assigned to constants with a **u** or **U** suffix is **unsigned int** or **unsigned long int** depending on their size.
- The type assigned to constants with an **l** or **L** suffix is **long int** or **unsigned long int** depending on their size.
- The type assigned to constants with a **u** or **U** and an **l** or **L** suffix is **unsigned long int**.

Integer Limits

Microsoft Specific

The limits for integer types are listed in Table 1.3. These limits are also defined in the standard header file LIMITS.H.

Table 1.3 Limits on Integer Constants

Constant	Meaning	Value
CHAR_BIT	Number of bits in the smallest variable that is not a bit field.	8
SCHAR_MIN	Minimum value for a variable of type **signed char**.	-127
SCHAR_MAX	Maximum value for a variable of type **signed char**.	127
UCHAR_MAX	Maximum value for a variable of type **unsigned char**.	255 (0xff)
CHAR_MIN	Minimum value for a variable of type **char**.	Same as -127; 0 if /J option used.
CHAR_MAX	Maximum value for a variable of type **char**.	Same as 127; 255 if /J option used.

Table 1.3 Limits on Integer Constants (*continued*)

Constant	Meaning	Value
MB_LEN_MAX	Maximum number of bytes in a multicharacter constant.	2
SHRT_MIN	Minimum value for a variable of type **short**.	-32767
SHRT_MAX	Maximum value for a variable of type **short**.	32767
USHRT_MAX	Maximum value for a variable of type **unsigned short**.	65535 (0xffff)
INT_MIN[1]	Minimum value for a variable of type **int**.	-32767
INT_MAX[2]	Maximum value for a variable of type **int**.	32767
UINT_MAX[3]	Maximum value for a variable of type **unsigned int**.	65535 (0xffff)
LONG_MIN	Minimum value for a variable of type **long**.	-2147483647
LONG_MAX	Maximum value for a variable of type **long**.	2147483647
ULONG_MAX	Maximum value for a variable of type **unsigned long**.	4294967295 (0xffffffff)

[1] The value for **INT_MIN** is–2147483648 for 32-bit target compilations.

[2] The value for **INT_MAX** is 2147483647 for 32-bit target compilations.

[3] The value for **UINT_MAX** is 4294967295 (0xffffffff)for 32-bit target compilations.

If a value exceeds the largest integer representation, the Microsoft compiler generates an error. ♦

Character Constants

A "character constant" is formed by enclosing a single character from the representable character set within single quotation marks (' '). Character constants are used to represent characters in the execution character set.

Syntax

character-constant :
  'c-char-sequence'
  L*'c-char-sequence'*

c-char-sequence :
  c-char
  c-char-sequence c-char

c-char :
 Any member of the source character set except
 the single quotation mark ('), backslash (\), or
 newline character
  escape-sequence

escape-sequence :
 simple-escape-sequence
 octal-escape-sequence
 hexadecimal-escape-sequence

simple-escape-sequence : one of
 \a \b \f \n \r \t \v
 \' \" \\ \?

octal-escape-sequence :
 \ *octal-digit*
 \ *octal-digit octal-digit*
 \ *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence :
 \x *hexadecimal-digit*
 hexadecimal-escape-sequence hexadecimal-digit

Character Types

An integer character constant not preceded by the letter **L** has type **int**. The value of an integer character constant containing a single character is the numerical value of the character interpreted as an integer. For example, the numerical value of the character `a` is 97 in decimal and 61 in hexadecimal.

Syntactically, a "wide-character constant" is a character constant prefixed by the letter **L**. A wide-character constant has type **wchar_t**, an integer type defined in the STDDEF.H header file. For example:

```
char    schar = 'x';   /* A character constant         */
wchar_t wchar = L'x';  /* A wide-character constant for
                          the same character           */
```

Wide-character constants are 16 bits wide and specify members of the extended execution character set. They allow you to express characters in alphabets that are too large to be represented by type **char**. See "Multibyte and Wide Characters" on page 8 for more information about wide characters.

Execution Character Set

This manual often refers to the "execution character set." The execution character set is not necessarily the same as the source character set used for writing C programs. The execution character set includes all characters in the source character set as well as the null character, newline character, backspace, horizontal tab, vertical tab, carriage return, and escape sequences. The source and execution character sets may differ in other implementations.

Escape Sequences

Character combinations consisting of a backslash (\) followed by a letter or by a combination of digits are called "escape sequences." To represent a newline character, single quotation mark, or certain other characters in a character constant, you must use escape sequences. An escape sequence is regarded as a single character and is therefore valid as a character constant.

Escape sequences are typically used to specify actions such as carriage returns and tab movements on terminals and printers. They are also used to provide literal representations of nonprinting characters and characters that usually have special meanings, such as the double quotation mark ("). Table 1.4 lists the ANSI escape sequences and what they represent.

Table 1.4 Escape Sequences

Escape Sequence	Represents
\a	Bell (alert)
\b	Backspace
\f	Formfeed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash
\?	Literal question mark
\*ooo*	ASCII character in octal notation
\x*hhh*	ASCII character in hexadecimal notation

Note that the question mark preceded by a backslash (**\?**) specifies a literal question mark in cases where the character sequence would be misinterpreted as a trigraph. See "Trigraphs" on page 8 for more information.

Microsoft Specific If a backslash precedes a character that does not appear in Table 1.4, the compiler handles the undefined character as the character itself. For example, `\x` is treated as an `x`.♦

Escape sequences allow you to send nongraphic control characters to a display device. For example, the ESC 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 formfeed escape sequences (**\v** and **\f**) do not affect screen output, but they do perform appropriate printer operations.

You can also use the backslash (\) as a continuation character. When a newline character (equivalent to pressing the RETURN key) immediately follows the backslash, the compiler ignores the backslash and the newline character and treats the next line as part of the previous line. This is useful primarily for preprocessor definitions longer than a single line. For example:

```
#define assert(exp) \
( (exp) ? (void) 0:_assert( #exp, __FILE__, __LINE__ ) )
```

In previous versions of the compiler, this feature was also used to create strings longer than one line. However, the string-concatenation feature (see "String Literals" on page 20) is now preferable when creating long string literals.

Octal and Hexadecimal Character Specifications

The sequence *ooo* means you can specify any character in the ASCII character set as a three-digit octal character code. The numerical value of the octal integer specifies the value of the desired character or wide character.

Similarly, the sequence **\x***hhh* allows you to specify any ASCII character as a hexadecimal character code. For example, you can give the ASCII backspace character as the normal C escape sequence (**\b**), or you can code it as **\010** (octal) or **\x008** (hexadecimal).

You can use only the digits 0 through 7 in an octal escape sequence. Octal escape sequences can never be longer than three digits and are terminated by the first character that is not an octal digit. Although you do not need to use all three digits, you must use at least one. For example, the octal representation is **\10** for the ASCII backspace character and **\101** for the letter A, as given in an ASCII chart.

Similarly, you must use at least one digit for a hexadecimal escape sequence, but you can omit the second and third digits. Therefore you could specify the hexadecimal escape sequence for the backspace character as either **\x8**, **\x08**, or **\x008**.

The value of the octal or hexadecimal escape sequence must be in the range of representable values for type **unsigned char** for a character constant and type

wchar_t for a wide-character constant. See "Multibyte and Wide Characters" on page 8 for information on wide-character constants.

Unlike octal escape constants, there is no limit on the number of hexadecimal digits in an escape sequence. A hexadecimal escape sequence terminates at the first character that is not a hexadecimal digit. Because hexadecimal digits include the letters **a** through **f**, care must be exercised to make sure the escape sequence terminates at the intended digit. To avoid confusion, you can place octal or hexadecimal character definitions in a macro definition:

```
#define Bell '\x07'
```

For hexadecimal values, you can break the string to show the correct value clearly:

```
"\xabc"     /* one character  */
"\xab" "c" /* two characters */
```

1.5 String Literals

A "string literal" is a sequence of characters from the source character set enclosed in double quotation marks (" "). String literals are used to represent a sequence of characters, which, taken together, form a null-terminated string. You must always prefix wide-string literals with the letter **L**.

Syntax

string-literal :
  "*s-char-sequence* $_{\text{opt}}$"
  L"*s-char-sequence* $_{\text{opt}}$"

s-char-sequence :
  s-char
  s-char-sequence s-char

s-char :
  any member of the source character set except the double quotation mark ("), backslash (\), or newline character
  escape-sequence

The example below is a simple string literal:

```
char amessage = "This is a string literal."
```

All escape codes listed in Table 1.4 are valid in string literals. To represent a double quotation mark in a string literal, use the escape sequence \". The single quotation mark (') can be represented without an escape sequence. The backslash (\) must be followed with a second backslash (\\) when it appears within a string.

When a backslash appears at the end of a line, it is always interpreted as a line-continuation character.

Type for String Literals

String literals have type array of **char** (that is, **char[ ]**). (Wide-character strings have type array of **wchar_t** (that is, **wchar_t[ ]**).) 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.

Storage of String Literals

The characters of a literal string are stored in order at contiguous memory locations. An escape sequence (such as \\ or \") within a string literal counts as a single character. A null character (represented by the **\0** escape sequence) is automatically appended to, and marks the end of, each string literal. (This occurs during translation phase 7, which is described on page 190.) Note also that the compiler may not store two identical strings at two different addresses.

String Literal Concatenation

To form string literals that take up more than one line, you can concatenate the two strings. To do this, type a backslash, then press the RETURN key. The backslash causes the compiler to ignore the following newline 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."
```

String concatenation can be used anywhere you might previously have used a backslash followed by a newline character to enter strings longer than one line. Because strings can start in any column of the source code without affecting their on-screen representation, you can position strings to enhance source-code readability.

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

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

Long strings can be continued in any column of a succeeding line without affecting their appearance when output. For example:

```
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, the parts are concatenated and output as a single string. This concatenation occurs according to the sequence of events during compilation specified by translation phases. See page 190 for information on translation phases.

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

A string pointer, initialized as two distinct string literals separated only by white space, is stored as a single string (pointers are discussed in "Pointer Declarations" on page 76). When properly referenced, as in the following example, the result is identical to the previous example:

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

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

In translation phase 6, the multibyte-character sequences specified by any sequence of adjacent string literals or adjacent wide-string literals are concatenated into a single multibyte-character sequence. Therefore, do not design programs to allow modification of string literals during execution. The ANSI standard specifies that the result of modifying a string is undefined.

Storage Class for Strings

Microsoft Specific Strings have static storage duration. See "Storage Classes" on page 43 for information about storage duration. ♦

Maximum String Length

Microsoft Specific ANSI compatibility requires a compiler to accept up to 509 characters in a string literal after concatenation. The maximum length of a string literal allowed with Microsoft C is 4,096 bytes. ♦

1.6 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. See page 1 for information on tokens.

Table 1.5 gives the precedence order for the C operators. For a complete description and the ANSI grammar for each operator, see "Operators" on page 111.

Table 1.5 Precedence and Associativity of C Operators

Symbol [1]	Type of Operation	Associativity
[] () . –> postfix ++ and postfix – – :>	Expression	Left to right
Prefix ++ and prefix – – **sizeof & * + – ~ !**	Unary	Right to left
typecasts	Unary	Right to left
* / %	Multiplicative	Left to right
+ –	Additive	Left to right
<< >>	Bitwise shift	Left to right
< > <= >=	Relational	Left to right
== !=	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[2]	Right to left
,	Sequential evaluation	Left to right

[1] Operators are listed in descending order of precedence. If several operators appear in the same line or in a group, they have equal precedence.

[2] All simple- and compound-assignment operators have equal precedence.

1.7 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 the compiled program carries out. They do not specify an operation to be performed. Some punctuation symbols are also operators (see the previous section). The compiler determines their use from context.

Syntax

punctuator : one of

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

These characters have special meanings in C. Their uses are described throughout this manual. The pound sign (#) can occur only in preprocessing directives. See "Manifest Constants and Macros" on page 191 for information about preprocessing directives.

Program Structure

Chapter 2

This chapter gives an overview of C programs and program execution. Terms and features important to understanding C programs and components are also introduced. After a brief look at program components, this chapter describes the **main** function and its arguments. Topics discussed include:

- Source files and translation units.
- The **main** function and command-line arguments.
- Lifetime and storage duration.
- Scope and visibility.
- Linkage.
- Name spaces.

Because this chapter is an overview, the topics discussed contain introductory material only. See the cross-referenced information for more detailed explanations.

2.1 Source Files and Source Programs

A source program can be divided into one or more "source files," or "translation units." The input to the compiler is called a "translation unit."

Syntax

translation-unit :
 external-declaration
 translation-unit external-declaration

external-declaration :
 function-definition
 declaration

"Overview of Declarations" on page 41 gives the syntax for the *declaration* nonterminal, and "Phases of Translation" on page 190 explains how the translation unit is processed.

Note See the introduction to Appendix B for an explanation of the ANSI grammar conventions.

The components of a translation unit are external declarations that include function definitions and identifier declarations. These declarations and definitions can be in source files, header files, libraries, and other files the program needs. You must compile each translation unit and link the resulting object files to make a program.

A C "source program" is a collection of directives, pragmas, declarations, definitions, statement blocks, and functions. To be valid components of a Microsoft C program, each must have the syntax described in this manual, although they can appear in any order in the program (subject to the rules outlined throughout this manual). However, the location of these components in a program does affect how variables and functions can be used in a program. (See "Understanding Lifetime, Scope, Visibility, and Linkage" on page 34 for more information.)

Source files 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 update when necessary. For the same reason, constants and macros are often organized into separate files called "include files" or "header files" that can be referenced in source files as required. ("Manifest Constants and Macros" on page 191 explains macros. See page 200 for information about include files.)

Preprocessor Directives

A "directive" instructs the C preprocessor to perform a specific action on the text of the program before compilation. Preprocessor directives are fully described in Chapter 7, "Preprocessor Directives and Pragmas." This example uses the preprocessor directive **#define**:

```
#define MAX 100
```

This statement tells the compiler to replace each occurrence of `MAX` by `100` prior to compilation. The C compiler preprocessor directives are

#define	**#endif**	**#ifdef**	**#line**
#elif	**#error**	**#ifndef**	**#pragma**
#else	**#if**	**#include**	**#undef**

Pragmas

A "pragma" instructs the compiler to perform a particular action at compile time. Pragmas vary from compiler to complier. For example, you can use the **optimize** pragma to set the optimizations to be performed on your program.

Microsoft Specific The Microsoft C pragmas are

alloc_text	**data_seg**	**linesize**	**pagesize**
auto_inline	**function**	**message**	**skip**
check_pointer	**hdrstop**	**native_caller**	**subtitle**
check_stack	**inline_depth**	**optimize**	**title**
code_seg	**inline_recursion**	**pack**	**warning**
comment	**intrinsic**	**page**	

These pragmas are described in "Pragma Directives" on page 209.

Declarations and Definitions

A "declaration" establishes an association between a particular variable, function, or type and its attributes. "Overview of Declarations" on page 41 gives the ANSI syntax for the *declaration* nonterminal. A declaration also specifies where and when an identifier can be accessed (the "linkage" of an identifier). See "Understanding Lifetime, Scope, Visibility, and Linkage" on page 34 for information about linkage.

A "definition" of a variable establishes the same associations as a declaration but also causes storage to be allocated for the variable.

For example, the `main`, `find`, and `count` functions and the `var` and `val` variables are defined in one source file, in this order:

```
main()
{
}

int var = 0;
double val[MAXVAL];

char find( fileptr )
{
}

int count( double f )
{
}
```

The variables `var` and `val` can be used in the `find` and `count` functions; no further declarations are needed. But these names are not visible (cannot be accessed) in `main`.

Function Declarations and Definitions

Function prototypes establish the name of the function, its return type, and the type and number of its formal parameters. A function definition includes the function body.

Both function and variable declarations can appear inside or outside a function definition. Any declaration within a function definition is said to appear at the "internal" or "local" level. A declaration outside all function definitions is said to appear at the "external," "global," or "file scope" level. 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. Function definitions are discussed further in "Function Definitions" on page 166. Function prototypes are covered in "Function Declarations" on page 84 and in "Function Prototypes" on page 181.

Blocks

A sequence of declarations, definitions, and statements enclosed within curly braces ({ }) is called a "block." There are two types of blocks in C. The "compound statement," a statement composed of one or more statements (discussed more fully on page 153), is one type of block. The other, the "function definition," consists of a compound statement (the body of the function) plus the function's associated "header" (the function name, return type, and formal parameters). A block within other blocks is said to be "nested."

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

Example Program

The following C source program consists of two source files. It gives an overview of some of the various declarations and definitions possible in a C program. Later sections in this manual describe how to write these declarations, definitions, and initializations, and how to use C keywords such as **static** and **extern**. The **printf** function is declared in the C header file STDIO.H.

The `main` and `max` functions are assumed to be in separate files, and execution of the program begins with the `main` function. No explicit user functions are executed before `main`.

```
/*********************************************************************
                     FILE1.C - main function
*********************************************************************/
```

```
#define ONE     1
#define TWO     2
#define THREE   3
#include <stdio.h>

int a = 1;                         /* Defining declarations     */
int b = 2;                         /*  of external variables    */

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

int main()                         /* Function definition       */
{                                  /*  for main function        */
    int c;                         /* Definitions for           */
    int d;                         /*  two uninitialized        */
                                   /*  local variables          */

    extern int u;                  /* Referencing declaration   */
                                   /*  of external variable     */
                                   /*  defined elsewhere        */
    static int v;                  /* Definition of variable    */
                                   /*  with continuous lifetime */

    int w = ONE, x = TWO, y = THREE;
    int z = 0;
    z = max( x, y );               /* Executable statements     */
    w = max( z, w );
    printf( "%d %d\n", z, w );
}

 /*****************************************************************
            FILE2.C - definition of max function
 *****************************************************************/

int max( int a, int b )            /* Note formal parameters are   */
                                   /*  included in function header */
{
    if( a > b )
        return( a );
    else
        return( b );
}
```

FILE1.C contains the prototype for the `max` function. This kind of declaration is sometimes called a “forward declaration” because the function is declared before it is used. The definition for the `main` function includes calls to `max`.

The lines beginning with `#define` are preprocessor directives. These directives tell the preprocessor to replace the identifiers `ONE`, `TWO`, and `THREE` with the numbers `1`, `2`, and `3`, respectively, throughout FILE1.C. However, the directives do not apply to FILE2.C, which is compiled separately and then linked with FILE1.C. The line beginning with `#include` tells the compiler to include the file STDIO.H, which contains the prototype for the **printf** function. Preprocessor directives are explained in Chapter 7.

FILE1.C uses defining declarations to initialize the global variables `a` and `b`. The local variables `c` and `d` are declared but not initialized. Storage is allocated for all these variables. The static and external variables, `u` and `v`, are automatically initialized to 0. Therefore only `a`, `b`, `u`, and `v` contain meaningful values when declared because they are initialized, either explicitly or implicitly. FILE2.C contains the function definition for `max`. This definition satisfies the calls to `max` in FILE1.C.

The lifetime and visibility of identifiers are discussed in "Understanding Lifetime, Scope, Visibility, and Linkage" on page 34. For more information on functions, see Chapter 6.

2.2 The main Function and Program Execution

Every C program has a primary (main) function that 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 **main**, although it can terminate at other points in the program for a variety of reasons. At times, perhaps when a certain error is detected, you may want to force the termination of a program. To do so, use the **exit** function. See the *Run-Time Library Reference* manual for information on and an example using the **exit** function.

Functions within the source program 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. A function returns control to **main** when a **return** statement is executed or when the end of the function is reached.

Microsoft Specific For the Microsoft C compiler, the convention is that successful termination is equivalent to a return of **EXIT_SUCCESS**, which is 0.

You can declare any function, including **main**, to have parameters. The term "parameter" or "formal parameter" refers to the identifier that receives a value passed to a function. See "Parameters" on page 179 for information on passing arguments to 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 formal parameters to **main** so that it can receive arguments from the command line using this format:

main (int *argc*, **char** **argv*[ ], **char** **envp*[ ])

When you want to pass information to the **main** function, the parameters are traditionally named *argc* and *argv*, although the C compiler does not require these names. The types for *argc* and *argv* are defined by the C language. Traditionally, if a third parameter is passed to **main**, that parameter is named *envp*. The type for the *envp* parameter is mandated by ANSI, but the name is not. Examples later in this chapter show how to use these three parameters to access command-line arguments. The following sections explain these parameters.

Argument Description

The *argc* parameter in the **main** function is an integer specifying how many arguments are passed to the program from the command line. Since the program name is considered an argument, the value of *argc* is at least one.

The *argv* parameter is an array of pointers to null-terminated strings representing the program arguments. Each element of the array points to a string representation of an argument passed to **main**. (See page 74 for information about arrays.) The *argv* parameter can be declared as an array of pointers to type **char** (`char *argv[]`) or as a pointer to pointers to type **char** (`char **argv`). The first string (`argv[0]`) is the program name. The last pointer (`argv[argc]`) is **NULL**. (See **getenv** in the *Run-Time Library Reference* manual for an alternative method for getting environment variable information.)

The *envp* parameter is a pointer to an array of null-terminated strings that represent the values set in the user's environment variables. The *envp* parameter can be declared as an array of pointers to **char** (`char *envp[]`) or as a pointer to pointers to **char** (`char **envp`). The end of the array is indicated by a **NULL** pointer.

Expanding Wildcard Arguments (Microsoft Specific)

When running a C program, you can use either of the two DOS wildcards—the question mark (?) and the asterisk (*)—to specify filename and pathname arguments on the command line.

Command-line arguments are handled by a routine called **_setargv**, which by default does not expand wildcards into separate strings in the *argv* string array. You can replace the normal **_setargv** routine with a more powerful version of **_setargv** that does handle wildcards by linking with the SETARGV.OBJ file.

You can link with SETARGV.OBJ from within PWB by adding SETARGV.OBJ to the program list for your program. You must specify the complete path or put SETARGV.OBJ in the current directory. You must also disable the Extended Dictionary option by turning off No Extended Dictionary in Library in the Link Options dialog box. To link with SETARGV.OBJ outside PWB, use the /NOE linker option. For example:

```
cl typeit.c setargv /link /NOE
```

The wildcards are expanded in the same manner as in DOS commands. (See your DOS user's guide if you are unfamiliar with wildcards.) Enclosing an argument in double quotation marks (" ") suppresses the wildcard expansion. Within quoted arguments, you can represent quotation marks literally by preceding the double-quotation-mark character with a backslash (\). If no matches are found for the wildcard argument, the argument is passed literally.

Parsing Command-Line Arguments (Microsoft Specific)

Microsoft C startup code uses the following rules when interpreting arguments given on the DOS command line:

- Arguments are delimited by white space, which is either a space or a tab.
- A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument. Note that the caret (^) is not recognized as an escape character or delimiter.
- A double quotation mark preceded by a backslash, \", is interpreted as a literal double quotation mark (").
- Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
- If an even number of backslashes is followed by a double quotation mark, then one backslash (\) is placed in the *argv* array for every pair of backslashes (\\), and the double quotation mark (") is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quotation mark, then one backslash (\) is placed in the *argv* array for every pair of backslashes (\\) and the double quotation mark is interpreted as an escape sequence by the remaining backslash, causing a literal double quotation mark (") to be placed in *argv*.

This list illustrates the rules above by showing the interpreted result passed to `argv` for several examples of command-line arguments. The output listed in the second, third, and fourth columns is from the ARGS.C program that follows the table.

Command-Line Input	**argv[1]**	**argv[2]**	**argv[3]**
`"a b c" d e`	`a b c`	`d`	`e`
`"ab\c" "\\" d`	`ab"c`	`\`	`d`
`a\\\b d"e f"g h`	`a\\\b`	`de fg`	`h`
`a\\\"b c d`	`a\"b`	`c`	`d`
`a\\\\"b c" d e`	`a\\b c`	`d`	`e`

```
/* ARGS.C illustrates the following variables used for accessing
 * command-line arguments and environment variables:
 * argc  argv  envp
 */

#include <stdio.h>

void main( int argc, /* Number of strings in array argv */
 char *argv[],       /* Array of command-line argument strings */
 char **envp )       /* Array of environment variable strings */
{
     int count;

    /* Display each command-line argument. */
    printf( "\nCommand-line arguments:\n" );
    for( count = 0; count < argc; count++ )
        printf( "  argv[%d]   %s\n", count, argv[count] );

    /* Display each environment variable. */
    printf( "\nEnvironment variables:\n" );
    while( *envp != NULL )
        printf( "  %s\n", *(envp++) );

    return;
}
```

One example of output from this program is:

```
Command-line arguments:
  argv[0]   C:\C700\TEST.EXE

Environment variables:
  COMSPEC=C:\DOS\COMMAND.COM

  PATH=c:\dos;c:\binb;c:\binr;c:\win3;c:\word;c:\help;c:\;
  PROMPT=[$p]
  TERM=vt52
  TEMP=c:\tmp
  TMP=c:\tmp
  EDITORS=c:\binr
  HOME=c:\lm.dos\netprog
```

Environment variables are set in your AUTOEXEC.BAT file.

Customizing Command-Line Processing

If your program does not take command-line arguments, you can save a small amount of space by suppressing use of the library routine that performs command-line processing. This routine is called **_setargv**, as described in "Expanding Wild-card Arguments" on page 31. To suppress its use, define a routine that does nothing in the file containing the **main** function and name it **_setargv**. The call to **_setargv** is then satisfied by your definition of **_setargv**, and the library version is not loaded.

Similarly, if you never access the environment table through the *envp* argument, you can provide your own empty routine to be used in place of **_setenvp**, the environment-processing routine.

If your program makes calls to the **_spawn** or **_exec** family of routines in the C run-time library, you should not suppress the environment-processing routine, since this routine is used to pass an environment from the parent process to the child process.

2.3 Understanding Lifetime, Scope, Visibility, and Linkage

To understand how a C program works, you must understand the rules that determine how variables and functions can be used in the program. Several concepts are crucial to understanding these rules:

- Lifetime
- Scope
- Visibility
- Linkage

Lifetime

"Lifetime" is the period during execution of a program in which a variable or function exists. The storage duration of the identifier determines its lifetime.

An identifier declared with the *storage-class-specifier* **static** has static storage duration. Identifiers with static storage duration (also called "global") have storage and a defined value for the duration of a program. Storage is reserved and the identifier's stored value is initialized only once, prior to program startup. An identifier declared with external or internal linkage also has static storage duration (see "Linkage" on page 36).

An identifier declared without the *storage-class-specifier* **static** has automatic storage duration if it is declared inside a function. An identifier with automatic storage duration (a "local identifier") has storage and a defined value only within the block where the identifier is defined or declared. An automatic identifier is allocated new storage each time the program enters that block, and it loses its storage (and its value) when the program exits the block. Identifiers declared in a function with no linkage also have automatic storage duration.

The following rules specify whether an identifier has global (static) or local (automatic) lifetime:

- All functions have static lifetime. Therefore they exist at all times during program execution. Identifiers declared at the external level (that is, outside all blocks in the program at the same level of function definitions) always have global (static) lifetimes.
- If a local variable has an initializer, the variable is initialized each time it is created (unless it is declared as **static**). Function parameters also have local lifetime. You can specify global lifetime for an identifier within a block by including the **static** *storage-class-specifier* in its declaration. Once declared **static**, the variable retains its value from one entry of the block to the next.

Although an identifier with a global lifetime 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. See the next section for information about visibility, and see page 43 for a discussion of the *storage-class-specifier* nonterminal.

Memory can be allocated as needed (dynamic) if created through the use of special library routines such as **malloc**. Since dynamic memory allocation uses library routines, it is not considered part of the language. See the **malloc** function in the *Run-Time Library Reference.*

Scope and Visibility

An identifier's "visibility" determines the portions of the program in which it can be referenced—its "scope." An identifier is visible (i.e., can be used) only in portions of a program encompassed by its "scope," which may be limited (in order of increasing restrictiveness) to the file, function, block, or function prototype in which it appears. The scope of an identifier is the part of the program in which the name can be used. This is sometimes called "lexical scope." There are four kinds of scope: function, file, block, and function prototype.

All identifiers except labels have their scope determined by the level at which the declaration occurs. The following rules for each kind of scope govern the visibility of identifiers within a program:

File scope

The declarator or type specifier for an identifier with file scope appears outside any block or list of parameters and is accessible from any place in the translation unit after its declaration. Identifier names with file scope are often called "global" or "external." The scope of a global identifier begins at the point of its definition or declaration and terminates at the end of the translation unit.

Function scope
  A label is the only kind of identifier that has function scope. A label is declared implicitly by its use in a statement. Label names must be unique within a function. (See page 157 for more information about labels and label names.)

Block scope
  The declarator or type specifier for an identifier with block scope appears inside a block or within the list of formal parameter declarations in a function definition. It is visible only from the point of its declaration or definition to the end of the block containing its declaration or definition. Its scope is limited to that block and to any blocks nested in that block and ends at the curly brace that closes the associated block. Such identifiers are sometimes called "local variables."

Function-prototype scope
  The declarator or type specifier for an identifier with function-prototype scope appears within the list of parameter declarations in a function prototype (not part of the function declaration). Its scope terminates at the end of the function declarator.

The appropriate declarations for making variables visible in other source files are described in "Storage Classes" on page 43. However, variables and functions declared at the external level with the **static** *storage-class-specifier* are visible only within the source file in which they are defined. All other functions are globally visible.

Linkage

Identifier names can refer to different identifiers in different scopes. An identifier declared in different scopes or in the same scope more than once can be made to refer to the same identifier or function by a process called "linkage." Linkage determines the portions of the program in which an identifier can be referenced (its "visibility"). There are three kinds of linkage: internal, external, and none.

Internal Linkage

If the declaration of a file-scope identifier for an object or a function contains the *storage-class-specifier* **static**, the identifier has internal linkage. Otherwise, the identifier has external linkage. See page 43 for a discussion of the *storage-class-specifier* nonterminal.

Within one translation unit, each instance of an identifier with internal linkage denotes the same identifier or function. Internally linked identifiers are unique to a translation unit.

External Linkage

If the first declaration at file-scope level for an identifier does not use the **static** storage-class specifier, the object has external linkage.

If the declaration of an identifier for a function has no *storage-class-specifier*, its linkage is determined exactly as if it were declared with the *storage-class-specifier* **extern**. If the declaration of an identifier for an object has file scope and no *storage-class-specifier*, its linkage is external.

An identifier's name with external linkage designates the same function or data object as does any other declaration for the same name with external linkage. The two declarations can be in the same translation unit or in different translation units. If the object or function also has global lifetime, the object or function is shared by the entire program.

No Linkage

If a declaration for an identifier within a block does not include the **extern** storage-class specifier, the identifier has no linkage and is unique to the function.

The following identifiers have no linkage:

- An identifier declared to be anything other than an object or a function
- An identifier declared to be a function parameter
- A block-scope identifier for an object declared without *storage-class-specifier* **extern**

If an identifier has no linkage, declaring the same name again (in a declarator or type specifier) in the same scope level generates a symbol redefinition error.

Summary

Table 2.1 is a summary of lifetime and visibility characteristics for most identifiers. The first three columns give the attributes that define lifetime and visibility. An identifier with the attributes given by the first three columns has the lifetime and visibility shown in the fourth and fifth columns. However, the table does not cover all possible cases. Refer to the discussion "Storage Classes" on page 43 for more information.

Table 2.1 Summary of Lifetime and Visibility

Attributes:			Result:	
Level	**Item**	**Storage-Class Specifier**	**Lifetime**	**Visibility**
File scope	Variable definition	**static**	Global	Restricted to remainder of source file in which it occurs
	Variable declaration	**extern**	Global	Remainder of source file
	Function prototype or definition	**static**	Global	Restricted to single source file
	Function prototype	**extern**	Global	Remainder of source file
Block scope	Variable declaration	**extern**	Global	Block
	Variable definition	**static**	Global	Block
	Variable definition	**auto** or **register**	Local	Block

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

```
#include <stdio.h>

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

int main()              /* main function defined at external level */
{
    printf( "%d\n", i ); /* Prints 1 (value of external level i)   */
    {                           /* Begin first nested block         */
        int i = 2, j = 3;       /* i and j defined at internal level */
        printf( "%d\n%d\n", i, j );    /* Prints 2, 3               */
        {                       /* Begin second nested block        */
            int i = 0;          /* i is redefined:                  */
            printf( "%d\n%d\n", i, j ); /* Prints 0, 3:             */
        }                       /* End of second nested block       */
        printf( "%d\n", i );    /* Prints 2 (outer definition       */
                                /*  restored):                      */
    }                           /* End of first nested block        */
    printf( "%d\n", i );        /* Prints 1 (external level         */
                                /* definition restored):            */
    return 0;
}
```

In this example, there are four levels of visibility: the external level and three block levels. The values are printed to the screen as noted in the comments following each statement.

2.4 Name Spaces

The compiler sets up "name spaces" to distinguish between the identifiers used for different kinds of items. The names within each name space must be unique to avoid conflict, but an identical name can appear in more than one name space. This means that you can use the same identifier for two or more different items, provided that the items are in different name spaces. The compiler can resolve references based on the syntactic context of the identifier in the program.

This list describes the name spaces used in C.

Statement labels
: Named statement labels are part of statements. Definitions of statement labels are always followed by a colon but are not part of **case** labels. Uses of statement labels always immediately follow the keyword **goto**. Statement labels do not have to be distinct from other names or from label names in other functions.

Structure, union, and enumeration tags
: These tags are part of structure, union, and enumeration type specifiers and, if present, always immediately follow the reserved words **struct**, **union**, or **enum**. The tag names must be distinct from all other structure, enumeration, or union tags with the same visibility.

Members of structures or unions
: Member names are allocated in name spaces associated with each structure and union type. That is, the same identifier can be a component name in any number of structures or unions at the same time. Definitions of component names always occur within structure or union type specifiers. Uses of component names always immediately follow the member-selection operators (**–>** and **.**). The name of a member must 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, or the name of the structure itself.

Ordinary identifiers
: All other names fall into a name space that includes variables, functions (including formal parameters and local variables), and enumeration constants. Identifier names have nested visibility, so you can redefine them within blocks.

Typedef names
: Typedef names cannot be used as identifiers in the same scope.

For example, since structure tags, structure members, and variable names are in three different name spaces, the three items named `student` in this example do not conflict. The context of each item allows correct interpretation of each occurrence of `student` in the program. (See page 65 for information about structures.)

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

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. However, overloading the tag name space is not recommended since it obscures meaning.

Declarations and Types

Chapter 3

This chapter describes the declaration and initialization of variables, functions, and types. The C language includes a standard set of basic data types. You can also add your own data types, called "derived types," by declaring new ones based on types already defined. Topics discussed include

- Storage-class specifiers
- Type specifiers
- Type qualifiers
- Declarators and variable declarations
- Initializers
- Enumeration, structure, union, and array declarations
- Pointers and based pointers
- Storage of basic types
- Typedef declarations

3.1 Overview of Declarations

A "declaration" specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for the object or function named by the identifier is called a "definition." C declarations for variables, functions, and types have this syntax.

Syntax

declaration :
  declaration-specifiers init-declarator-list $_{\text{opt}}$;

declaration-specifiers :
  storage-class-specifier declaration-specifiers $_{\text{opt}}$
  type-specifier declaration-specifiers $_{\text{opt}}$
  type-qualifier declaration-specifiers $_{\text{opt}}$

init-declarator-list :
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator :
  declarator
  declarator = *initializer*

Note This syntax for *declaration* is not repeated in the following sections. Syntax in the following sections usually begin with the *declarator* nonterminal.

The declarations in the *init-declarator-list* contain the identifiers being named; *init* is an abbreviation for initializer. The *init-declarator-list* is a comma-separated sequence of declarators, each of which can have additional type information, or an initializer, or both. The *declarator* contains the identifiers, if any, being declared. The *declaration-specifiers* nonterminal consists of a sequence of type and storage-class specifiers that indicate the linkage, storage duration, and at least part of the type of the entities that the declarators denote. Therefore, declarations are made up of some combination of storage-class specifiers, type specifiers, type qualifiers, declarators, and initializers.

In the general form of a variable declaration, *type-specifier* gives the data type of the variable. The *type-specifier* can be a compound, as when the type is modified by **const**, **volatile**, or one of the special keywords described in "Special Keywords in Declarations" on page 55. The *declarator* gives the name of the variable, possibly modified to declare an array or a pointer type. For example,

```
int const __far *fp;
```

declares a variable named `fp` as a far pointer to a nonmodifiable (**const**) **int** value. You can define more than one variable in a declaration by using multiple declarators, separated by commas.

A declaration must have at least one declarator, or its type specifier must declare a structure tag, union tag, or members of an enumeration. Declarators provide any remaining information about an identifier. 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. "Declarators and Variable Declarations" on page 53 discusses declarators.

All definitions are implicitly declarations, but not all declarations are definitions. For example, variable declarations that begin with the **extern** storage-class specifier are "referencing," rather than "defining" declarations. If an external variable is to be referred to before it is defined, or if it is defined in another source file from the one where it is used, an **extern** declaration is necessary. Storage is not allocated by "referencing" declarations, nor can variables be initialized in declarations.

A storage class or a type (or both) is required in variable declarations. Only one storage-class specifier is allowed in a declaration and not all storage-class specifiers are permitted in every context. 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. The *storage-class-specifier* nonterminals defined in C include: **auto**, **extern**, **register**, **static**, and **typedef**. All the *storage-class-specifier* nonterminals except **typedef** are discussed in "Storage Classes" on page 43. See "Typedef Declarations" on page 101 for information about **typedef**.

The location of the declaration within the source program and the presence or absence of other declarations of the variable are important factors in determining the lifetime of variables. There can be multiple redeclarations but only one definition. However, a definition can appear in more than one translation unit. For objects with internal linkage, this rule applies separately to each translation unit, because internally linked objects are unique to a translation unit. For objects with external linkage, this rule applies to the entire program. See "Understanding Lifetime, Scope, Visibility, and Linkage" on page 34 for more information about visibility.

Type specifiers provide some information about the data types of identifiers. The default type specifier is **int**. Type specifiers are discussed in "Type Specifiers" on page 51. Type specifiers may also define type tags, structure and union component names, and enumeration constants. Enumerations, structures, and unions are discussed later in this chapter beginning on page 62.

There are two *type-qualifier* nonterminals: **const** and **volatile**. These qualifiers specify additional properties of types that are relevant only when accessing objects of that type through l-values. For more information on **const** and **volatile**, see "Type Qualifiers" on page 52. For a definition of l-values, see page 107.

3.2 Storage Classes

The "storage class" of a variable determines whether the item has a "global" or "local" lifetime. C calls these two lifetimes "static" and "automatic." An item with a global lifetime exists and has a value throughout the execution of the program. All functions have global lifetimes.

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

C provides the following storage-class specifiers:

Syntax

storage-class-specifier :
  auto
  register
  static
  extern
  typedef

At most one *storage-class-specifier* may be given in the *declaration-specifier* in a declaration. If no storage-class specification is made, declarations within a block create automatic objects.

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

Since **typedef** is semantically different from the other four *storage-class-specifier* nonterminals, it is discussed separately in "Typedef Declarations" on page 101.

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

"Storage-Class Specifiers for External-Level Declarations" on page 44 and "Storage-Class Specifiers for Internal-Level Declarations" on page 47 describe the *storage-class-specifier* nonterminals in each kind of declaration and explain the default behavior when the *storage-class-specifier* nonterminal is omitted from a variable. "Storage-Class Specifiers with Function Declarations" on page 50 discusses storage-class specifiers used with functions.

Storage-Class Specifiers for External-Level Declarations

External variables are variables at file scope. They are defined outside any function, and they are potentially available to many functions. Functions may only be defined at the external level and, therefore, cannot be nested. By default, all references to external variables and functions of the same name are references to the same object, which means they have "external linkage." (You can use the **static** keyword to override this. See information later in this section for more details on **static**.)

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 declaration of the variable. A definition 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 "Initialization" on page 91. If you omit the initializer, the variable is initialized to 0 by default. For example, these two statements are both considered definitions of the variable `k`.

  ```
  static int k = 16;
  static int k;
  ```

- A variable that you explicitly initialize at the external level. For example, `int j = 3;` is a definition of the variable `j`.

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-specifier* nonterminals at the external level.

Once a variable is defined at the external level, it is visible throughout the rest of the translation unit. The variable is not visible prior to its declaration 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.

The rules relating to **static** include:

- Variables declared outside all blocks without the **static** keyword always retain their values throughout the program. To restrict their access to a particular translation unit, you must use the **static** keyword. This gives them "internal linkage." To make them global to an entire program, omit the explicit storage class or use the keyword **extern** (see the rules in the next list). This gives them "external linkage." Internal and external linkage are also discussed in "Linkage" on page 36.
- You can define a variable at the external level only once within a program. You can define another variable with the same name and the **static** storage-class specifier in a different translation unit. Since each **static** definition is visible only within its own translation unit, no conflict occurs. This provides a useful way to hide identifier names that must be shared among functions of a single translation unit, but not visible to other translation units.

- The **static** storage-class specifier can apply to functions as well. If you declare a function **static**, its name is invisible outside of the file in which it is declared.

The rules for using **extern** are

- 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 prior to 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 translation unit in which the declared reference occurs.
- For an **extern** reference to be valid, the variable it refers to must be defined once, and only once, at the external level. This definition (without the **extern** storage class) can be in any of the translation units that make up the program.

The example below illustrates external declarations:

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

extern int i;                 /* Reference to i, defined below */
void next( void );            /* Function prototype            */

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

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

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

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

extern int i;                 /* Reference to i in  */
                              /* first source file  */
void other( void )
{
    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 is a "defining declaration." That declaration,

```
int i = 3
```

defines the global variable i and initializes it with initial value 3. The "referencing" declaration of i at the top of the first source file using **extern** makes the global variable visible prior to its defining declaration in the file. The referencing declaration of i in the second source file also makes the variable visible in that source file. If a defining instance for a variable is not provided in the translation unit, the compiler assumes there is an

```
extern int x;
```

referencing declaration and that a defining reference

```
int x = 0;
```

appears in another translation unit of the program.

All three functions, main, next, and other, 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. In this case, the values 1, 2, and 3 would have been printed. See "Initialization" on page 91 for information about variable initialization.

Storage-Class Specifiers for Internal-Level Declarations

You can use any of the four *storage-class-specifier* nonterminals for variable declarations at the internal level. When you omit the *storage-class-specifier* from such a declaration, the default storage class is **auto**. Therefore, the keyword **auto** is rarely seen in a C program.

The auto Storage-Class Specifier

The **auto** storage-class specifier declares an automatic variable, 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 "Initialization" on page 91. 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 local variable of **auto** or **register** storage class is initialized each time it comes in scope if an initializer is given.)

A internal **static** variable (a static variable with local or block scope) 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

The **register** *storage-class-specifier* tells the compiler to give an automatic 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. You cannot apply the unary address-of operator (**&**) to a register object (see page 111 for information about operators).

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. Register storage, if available, is only guaranteed for **int** and pointer types that are the same size as an **int**. (An initialized local variable of **automatic** or **register** storage class is initialized each time it comes in scope if an initializer is given.)

Microsoft Specific

The Microsoft C/C++ version 7.0 16-bit compiler uses the SI and DI registers for **register** variables.

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za command-line option to disable these extensions.♦

32-Bit Specific

The 32-bit compiler uses the ESI, EDI, and EBX registers.♦

The static Storage-Class Specifier

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. For constant strings, using **static** is useful because it alleviates the overhead of frequent initialization in often-called functions.

If you do not explicitly initialize a **static** variable, it is initialized to 0 by default. Inside a function, **static** causes storage to be allocated and serves as a definition. Internal static variables provide private, permanent storage visible to only a single function.

The extern Storage-Class Specifier

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. This example illustrates internal- and external-level declarations:

```
#include <stdio.h>

int i = 1;

void other( void );

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();
    return;
}

void other( void )
{
    /* 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 the other function: */
    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` 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 does not change during program execution. 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 does 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 reenters the block in which they are declared.

Storage-Class Specifiers with Function Declarations

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

Microsoft Specific 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 throughout the rest of the translation unit even if it is declared at local scope. ♦

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 access it directly by name. 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 are **extern** by default.

Microsoft Specific Microsoft allows redefinition of an **extern** identifier as **static**. ♦

3.3 Type Specifiers

Type specifiers in declarations define the type of a variable or function declaration.

Syntax

type-specifier :
  void
  char
  short
  int
  long
  float
  double
  signed
  unsigned
  struct-or-union-specifier
  enum-specifier
  typedef-name

The **signed char**, **signed int**, **signed short int**, and **signed long int** types, together with their **unsigned** counterparts and **enum**, are called "integral" types. The **float**, **double**, and **long double** type specifiers are referred to as "floating" or "floating-point" types. You can use any integral or floating-point type specifier in a variable or function declaration. If a *type-specifier* is not provided in a declaration, it is taken to be **int**.

The optional keywords **signed** and **unsigned** can precede or follow any of the integral types, except **enum**, and can also be used alone as type specifiers, in which case they are understood as **signed int** and **unsigned int**, respectively. 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**.

Enumeration types are considered basic types. Type specifiers for enumeration types are discussed in "Enumeration Declarations" on page 62.

The keyword **void** has three uses: to specify a function return type, to specify an argument-type list for a function that takes no arguments, and to specify a pointer to an unspecified type. You can use the **void** type to declare functions that return no value or to declare a pointer to an unspecified type. See "Arguments" on page 185 for information on **void** when it appears alone within the parentheses following a function name.

Type **void** expressions are evaluated for side effects. You cannot use the (nonexistent) value of an expression that has type **void** in any way, nor can you convert a **void** expression (by implicit or explicit conversion) to any type except **void**. If you do use an expression of any other type in a context where a **void** expression is required, its value is discarded.

You can create additional type specifiers with **typedef** declarations, as described in "Typedef Declarations" on page 101. See page 98 for information on the size of each type.

This manual generally uses the forms of the type specifiers listed in Table 3.1 rather than the long forms, and it assumes that the **char** type is signed by default. Therefore, throughout this manual, **char** is equivalent to **signed char**.

Table 3.1 Type Specifiers and Equivalents

Type Specifier	Equivalent(s)
signed char[1]	**char**
signed int	**signed**, **int**
signed short int	**short**, **signed short**
signed long int	**long**, **signed long**
unsigned char[1]	—
unsigned int	**unsigned**
unsigned short int	**unsigned short**
unsigned long int	**unsigned long**
float	—
long double	—

[1] When you make the **char** type unsigned by default (by specifying the /J compiler option), you cannot abbreviate **signed char** or **unsigned char** as **char**.

Microsoft Specific

You can specify the /J compiler option to change the default **char** type from signed to unsigned. When this option is in effect, **char** means the same as **unsigned char**, and you must use the **signed** keyword to declare a signed character value. The /J command-line option does not effect a **char** value that is explicitly declared signed. The **char** type is zero-extended, not sign-extended, when /J is specified. ♦

3.4 Type Qualifiers

Type qualifiers give one of two properties to an identifier. The **const** type qualifier declares an object to be nonmodifiable. The **volatile** type qualifier declares an item whose value can legitimately be changed by something beyond the control of the program in which it appears, such as a concurrently executing thread. The two type qualifiers, **const** and **volatile**, can appear only once in a declaration and must be placed after the type they qualify. Type qualifiers can appear with any type specifier. They are relevant only when accessing identifiers as l-values in expressions. See "L-Value and R-Value Expressions" on page 107 for information about l-values and expressions.

Syntax

type-qualifier :
  const
  volatile

The following are legal **const** and **volatile** declarations:

```
int const *p_ci;       /* Pointer to constant int */
int const (*p_ci);     /* Pointer to constant int */
int *const cp_i;       /* Constant pointer to int */
int (*const cp_i);     /* Constant pointer to int */
int volatile vint;     /* Volatile integer       */
```

If the specification of an array type includes type qualifiers, the element is qualified, not the array type. If the specification of the function type includes qualifiers, the behavior is undefined. Neither **volatile** nor **const** affect the range of values or arithmetic properties of the object.

This list describes how to use **const** and **volatile**.

- The **const** keyword can be used to modify any fundamental or aggregate type, a pointer to an object of any type, or a **typedef**. If an item is declared with only the **const** type qualifier, its type is taken to be **const int**. A **const** variable can be initialized. A **const** object can be placed in a read-only region of storage. The **const** keyword is useful for declaring pointers to **const** since this requires the function not to change the pointer in any way.
- The compiler assumes that, at any point in the program, a **volatile** variable can be accessed by an unknown process that uses or modifies its value. Therefore, regardless of the optimizations specified on the command line, the code for each assignment to or reference of a **volatile** variable must be generated even if it appears to have no effect.

  If **volatile** is used alone, **int** is assumed. The **volatile** type specifier can be used to provide reliable access to special memory locations. Use **volatile** with data objects that may be accessed or altered by signal handlers, by concurrently executing programs, or by special hardware such as memory-mapped I/O control registers. You can declare a variable as **volatile** for its lifetime, or you can cast a single reference to be **volatile**.
- An item can 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 asynchronous process.

3.5 Declarators and Variable Declarations

The rest of this chapter describes the form and meaning of declarations for variable types summarized in this list. In particular, the remaining sections explain how to declare the following:

Type of Variable	Description
Simple variables	Single-value variables with integral or floating-point type
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
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 can have different types
Unions	Variables composed of several values of different types that occupy the same storage space

A declarator is the part of a declaration that specifies the name that is to be introduced into the program. It can include modifiers such as * (pointer-to) and any of the Microsoft calling-convention and memory-model keywords.

Microsoft Specific

In the declarator

```
char * __far *var;
```

char is the type specifier, `*__far` and `*` are the modifiers, and `var` is the identifier's name.♦

You use declarators to declare arrays of values, pointers to values, and functions returning values of a specified type. Declarators appear in the pointer, array, and function declarations described later in this chapter.

Syntax

declarator :
  pointer $_{\text{opt}}$ *direct-declarator*

direct-declarator :
  identifier
  (*declarator*)
  direct-declarator [*constant-expression* $_{\text{opt}}$]
  direct-declarator (*parameter-type-list*)
  direct-declarator (*identifier-list* $_{\text{opt}}$)

pointer :
  * *type-qualifier-list* $_{\text{opt}}$
  * *type-qualifier-list* $_{\text{opt}}$ *pointer*

type-qualifier-list :
  type-qualifier
  type-qualifier-list type-qualifier

Note See the syntax for *declaration* in "Overview of Declarations" on page 41, or look in Appendix A for the syntax that references a *declarator.*

When a declarator consists of an unmodified identifier, the item being declared has a base type. If an asterisk (*) appears 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 type. See page 88 for more information about interpreting precedence within declarations.

Each declarator declares at least one identifier. 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.

Microsoft Specific

Up to 12 pointer, array, and function declarators in any valid combination are allowed to modify an arithmetic, structure, union, or incomplete type either directly or with **typedef** declarations. ♦

Array, pointer, and function declarations are discussed in more detail later in this chapter. The following examples illustrate a few simple forms of declarators:

```
int list[20]; /* Declares an array of int values named list */

char *cp;     /* Declares a pointer named cp to a char value */

double func( void ); /* Declares a function named func, with no
                        arguments, that returns a double value */

int *aptr[1]   /* Declares an array of 10 pointers */
```

Microsoft Specific

The Microsoft C compiler does not limit the number of declarators that can modify an arithmetic, structure, or union type. The number is limited only by available memory. ♦

Special Keywords in Declarations (Microsoft Specific)

The following special keywords can also be used in declarations. Since these keywords are not used in all C compilers, code using these keywords may not be portable.

- The special keywords related to addressing are **__near**, **__far**, **__huge**, and **__based**.
- The special keywords related to calling conventions are **__cdecl**, **__fortran**, **__pascal**, **__stdcall**, and **__fastcall**.

These keywords modify the meaning of variable and function declarations. 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 myvar` and `__pascal __far myvar` have the same effect). Thus the "binding" characteristics of the special keywords are the same as those of the type specifiers **const** and **volatile** except that **const** and **volatile** bind to the left and the keywords such as **__far** bind to the right. ("Type Qualifiers" on page 52 contains descriptions of the **const** and **volatile** keywords.)

Using Address-Related Keywords

The **__near**, **__far**, **__huge**, and **__based** keywords modify either objects or pointers to objects. If an object identifier is to the right of the keyword, the keyword determines if the object will be allocated in the default data segment or in a separate data segment. If a pointer is to the right of the keyword, the keyword determines whether the pointer will hold a near, based, far, or huge address. Examples using these keywords appear in "Using the Special Keywords" on page 59.

You can override the default addressing convention for a given function, function call, or data reference by declaring code or data items as near, far, huge, or based. You don't need to change the addressing conventions for the program as a whole if you use the **__near**, **__far**, **__huge**, or **__based** keyword.

The following list summarizes the characteristics of data defined with these keywords. See "Function Attributes" on page 168 for additional information on using these keywords in function declarations.

__near
Data resides in the default data segment, _DATA. It is referenced with 16-bit addresses (near pointers to data are 16 bits), so data addressed can be in the default segment only. Variables declared with the **__near** keyword use 16-bit pointer arithmetic.

32-Bit Specific

The 32-bit compiler does not allow the use of the **__near** keyword.♦

__far
Data can be anywhere in memory. It is not assumed to reside in the current data segment. Data items are referenced with 32-bit addresses (far pointers to data are 32 bits). Data objects declared as far must reside within the segment in which they start. Therefore, they must be smaller than 64K. Variables declared with the **__far** keyword use 16-bit pointer arithmetic.

32-Bit Specific The 32-bit compiler does not allow the use of the **__far** keyword.♦

__huge
Data can be anywhere in memory. It is not assumed to reside in the current data segment. Individual arrays can exceed 64K in size. Data is referenced with 32-bit addresses (huge pointers to data are 32 bits). The **__huge** keyword is not applicable to functions. Variables declared with the **__huge** keyword use 32-bit pointer arithmetic for data (same restrictions as for arrays specified with the **__huge** keyword). Objects declared as huge trade efficiency in pointer arithmetic for relaxed limits on array and object size. Objects with automatic storage class cannot be declared as huge.

32-Bit Specific The 32-bit compiler does not allow the use of the **__huge** keyword.♦

__based
Data can be anywhere in memory. It is not assumed to reside in the current data segment. The range of 32-bit addresses is provided by 16-bit addresses plus a programmer-provided base. A based pointer is a 16-bit value interpreted as an offset from a supplied base. Variables declared with the **__based** keyword use 16-bit pointer arithmetic for data. See "Based Pointers" on page 79 for more information on **__based**.

The **__based** keyword is recommended instead of the **alloc_text** pragma for specifying the location of a function. However, the **alloc_text** pragma is still supported.

32-Bit Specific For 32-bit targets, **__based** specifies that a pointer is a 32-bit offset from a 32-bit base.♦

Using Calling-Convention-Related Keywords

These modifiers are placed before the function name and can appear before or after the **__near**, **__far**, **__huge**, or **__based** modifiers. This list explains each of the keywords used to specify calling conventions. These keywords begin with two underscores. See "Identifiers" on page 5 for information about this ANSI naming convention.

__pascal, __fortran
Specifies that the associated function is to be called using the Pascal or FORTRAN calling convention (arguments are pushed from left to right). The **__fortran** and **__pascal** modifiers are synonyms.

32-Bit Specific The **__fortran** and **__pascal** keywords are not accepted for 32-bit targets.♦

__cdecl
Specifies that the associated function is to be called using the normal C calling convention (arguments are pushed from right to left). Use this specifier with individual functions if the /Gc compiler option may have been set to make the

Pascal/FORTRAN calling convention the default. This is the default for 16-bit targets.

32-Bit Specific

The **__cdecl** calling convention is the default for 32-bit targets.♦

__fastcall
Specifies that the function uses a calling convention that passes arguments in registers rather than on the stack, resulting in faster code. The **__fastcall** calling convention cannot be used with functions having variable-length parameter lists, or functions having any of the following attributes: **__export**, **__interrupt**, **__saveregs**. The 16-bit compiler uses the AX, BX, and DX registers, but Microsoft reserves the right to change the registers and implementation of the **__fastcall** calling convention between releases of the compiler.

32-Bit Specific

The 32-bit compiler uses the ECX and EDX registers, but Microsoft reserves the right to change the registers and implementation of the **__fastcall** calling convention between releases of the compiler.♦

32-Bit Specific

__stdcall
Specifies that the arguments of the designated function are pushed right to left, that an underscore is prepended to the name, and that an at-sign character (@) followed by the number of bytes in the argument list is appended to the name (called "name decoration"). The **__stdcall** calling convention is only available for 32-bit targets.♦

__export
Specifies that the compiler should place information in the object file to show that the symbol is exported from a dynamic-link library (DLL) to Microsoft Windows™.

Nonstatic variables defined with the **__pascal** keyword cannot be distinguished by case. For example, the following two variables have external linkage and the compiler resolves them to one variable:

```
int __pascal A;
int __pascal a;
```

However, the following two definitions have internal linkage and the compiler resolves them to two distinct variables:

```
static int __pascal A;
static int __pascal a;
```

See "Function Attributes" on page 168 for more information on function calling conventions.

Data Declarations with the __based Keyword

Static and external objects can be declared using the **__based** keyword. In this context, the **__based** specification causes the object to be allocated in the specified segment. See Chapter 4 of *Programming Techniques* for more information.

To specify the segment for storing data, you can use the built-in function **__segname**. This function accepts a string literal and returns a value of type **__segment**. The code looks like this:

__segname(*string-literal* **)**

This declares a based variable by giving a segment constant as a base. The *string-literal* can be the name of one of four predefined segments (_CODE, _CONST, _DATA, or _STACK), or it can be the name of a new segment you define.

External data based on *seg_expr*, a segment variable of type **__segment**, has the form

extern *type* **__based(** *seg_expr)*

Data declared this way resides in a location determined at run time. You can relocate a segment in memory, set *seg_expr* to the new location of that segment, and access a variable stored in that segment without using pointers.

Data based on the address of another variable has the form

__based((__segment)&*var* **)**

The *var* specified must itself be based on a named segment. This declaration places both variables in the same segment.

32-Bit Specific

The **__segment** and **__segname** keywords are not available with the 32-bit compiler.♦

Using the Special Keywords

This section provides examples of how to use the special keywords. You can 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 occurrences of the **__far** and **__near** keywords:

```
int __far * __pascal __near func( void );
```

In this example, the **__pascal** and **__near** 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 "Interpreting More Complex Declarators" on page 88) also apply to declarations that use the special keywords. The memory model determines the default size for pointers. This can be specified with the command-line options /AL, /AS, /AM, and /AH, or (within PWB) by selecting the memory model in the C Compiler Options dialog box.

```
int __huge database[65000];
```

This example declares a huge array named `database` with 65,000 **int** elements. The **__huge** keyword modifies the array declarator.

```
char * __far * x;
char * (__far *x);
```

In these statements, the **__far** keyword modifies the asterisk to its right, making `x` a far pointer to a pointer to **char**. The second statement is an alternative way to write this declaration that can make your code easier to read.

```
double __near __cdecl calc( double, double );

double __cdecl __near calc( double, double );
```

Since the special keywords can be used in any order, these two declarations are equivalent. Both declare `calc` as a function with the **__near** and **__cdecl** attributes.

```
char __far __fortran initlist[INITSIZE];

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

In the two declarations above, 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.

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

This example shows a more complex declaration with several occurrences of the **__far** keyword. The numbers indicate the order in which the declaration is interpreted in the following procedure:

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. "Interpreting More Complex Declarators" on page 88 provides more information about interpreting complex declarators.

Simple Variable Declarations

The declaration of a simple variable, the simplest form of a direct declarator, specifies the variable's name and type. It also specifies the variable's storage class and data type.

Storage classes or types (or both) are required on variable declarations. Untyped variables (such as `a;`) generate warnings.

Syntax

declarator :
  pointer $_{\text{opt}}$ *direct-declarator*

direct-declarator :
  identifier

identifier :
  nondigit
  identifier nondigit
  identifier digit

For arithmetic, structure, union, enumerations, and void types, and for types represented by **typedef** names, simple declarators may be used in a declaration since the type specifier supplies all the typing information. Pointer, array, and function types require more complicated declarators.

You can use a list of identifiers separated by commas (,) to specify several variables in the same declaration. All variables defined in the declaration have the same base type. For example:

```
int x, y;        /* Declares two simple variables of type int */
int const z = 1; /* Declares a constant value of type int */
```

The variables `x` and `y` can hold any value in the set defined by the **int** type for a particular implementation. The simple object `z` is initialized to the value 1 and is not modifiable.

If the declaration of `z` was for an uninitialized static variable or was at file scope, it would receive an initial value of 0, and that value would be unmodifiable.

```
unsigned long reply, flag; /* Declares two variables
                              named reply and flag     */
```

In this example, both the variables, `reply` and `flag`, have **unsigned long** type and hold unsigned integral values.

Enumeration Declarations

An enumeration consists of a set of named integer constants. An enumeration type declaration gives the name of the (optional) enumeration tag and defines the set of named integer identifiers (called the "enumeration set," "enumerator constants," "enumerators," or "members"). A variable with enumeration type stores one of the values of the enumeration set defined by that type.

Variables of **enum** type may be used in indexing expressions and as operands of all arithmetic and relational operators. Enumerations provide an alternative to the **#define** preprocessor directive with the advantages that the values can be generated for you and obey normal scoping rules.

In ANSI C, the expressions that define the value of an enumerator constant always have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value. An enumeration constant or a value of enumerated type can be used anywhere the C language permits an integer expression.

Syntax

enum-specifier :
  enum *identifier* $_{\text{opt}}$ { *enumerator-list* }
  enum *identifier*

The optional *identifier* names the enumeration type defined by *enumerator-list.* This identifier is often called the "tag" of the enumeration specified by the list. A type specifier of the form

enum *identifier* { *enumerator-list* }

declares *identifier* to be the tag of the enumeration specified by the *enumerator-list* nonterminal. The *enumerator-list* defines the "enumerator content." The *enumerator-list* is described in detail below.

If the declaration of a tag is visible, subsequent declarations that use the tag but omit *enumerator-list* specify the previously declared enumerated type. The tag must refer to a defined enumeration type, and that enumeration type must be in current scope. Since the enumeration type is defined elsewhere, the *enumerator-list* does not appear in this declaration. Declarations of types derived from enumerations and **typedef** declarations for enumeration types can use the enumeration tag before the enumeration type is defined.

Syntax

enumerator-list :
  enumerator
  enumerator-list , *enumerator*

enumerator :
  enumeration-constant
  enumeration-constant = *constant-expression*

enumeration-constant :
  identifier

Each *enumeration-constant* in an *enumeration-list* names a value of the enumeration set. By default, the first *enumeration-constant* is associated with the value 0. The next *enumeration-constant* in the list is associated with the value of (*constant-expression* + 1), unless you explicitly associate it with another value. The name of an *enumeration-constant* is equivalent to its value.

You can use *enumeration-constant* = *constant-expression* to override the default sequence of values. Thus, if *enumeration-constant* = *constant-expression* appears in the *enumerator-list*, the enumeration-constant is associated with the value given by *constant-expression.* The *constant-expression* must have **int** type and can be negative.

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, perhaps named `null` and `zero`, in the same set.
- The identifiers in the enumeration list must be distinct from other identifiers in the same scope with the same visibility, including ordinary variable names and identifiers in other enumeration lists.
- Enumeration tags obey the normal scoping rules. They must be distinct from other enumeration, structure, and union tags with the same visibility.

These examples illustrate enumeration declarations:

```
enum DAY            /* Defines an enumeration type    */
{
    saturday,       /* names day and declares a       */
    sunday = 0,     /* variable named workday with    */
    monday,         /* that type                      */
    tuesday,
    wednesday,      /* wednesday is associated with 3 */
    thursday,
    friday
} workday;
```

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.

In this example, a value from the set `DAY` is assigned to the variable `today`.

```
enum DAY today = wednesday;
```

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 `DAY` is necessary.

To explicitly assign an integer value to a variable of an enumerated data type, use a type cast:

```
workday = ( enum DAY ) ( day_value - 1 );
```

This cast is recommended in C but is not required.

```
enum BOOLEAN  /* Declares an enumeration data type called BOOLEAN */
{
    false,     /* false = 0, true = 1 */
    true
};

enum BOOLEAN end_flag, match_flag; /* Two variables of type BOOLEAN */
```

This declaration can also be specified as

```
enum BOOLEAN { false, true } end_flag, match_flag;
```

or as

```
enum BOOLEAN { false, true } end_flag;
enum BOOLEAN match_flag;
```

An example that uses these variables might look like this:

```
    if ( match_flag == false )
    {
     .
     .   /* statement */
     .
    }
    end_flag = true;
```

Unnamed enumerator data types can also be declared. The name of the data type is omitted, but variables can be declared. The variable `response` is a variable of the type defined:

```
enum { yes, no } response;
```

Structure Declarations

A "structure declaration" names a type and specifies a sequence of variable values (called "members" or "fields" of the structure) that can have different types. An optional identifier, called a "tag," gives the name of the structure type and can be used in subsequent references to the structure type. A variable of that structure type holds the entire sequence defined by that type. Structures in C are similar to the types known as "records" in other languages.

Syntax

struct-or-union-specifier:
  struct-or-union identifier $_{\text{opt}}$ { *struct-declaration-list* }
  struct-or-union identifier

struct-or-union :
  struct
  union

struct-declaration-list:
  struct-declaration
  struct-declaration-list struct-declaration

The structure content is defined to be

struct-declaration :
  specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
  type-specifier specifier-qualifier-list $_{\text{opt}}$
  type-qualifier specifier-qualifier-list $_{\text{opt}}$

struct-declarator-list:
  struct-declarator
  struct-declarator-list, *struct-declarator*

struct-declarator :
  declarator

The declaration of a structure type does not set aside space for a structure. It is only a template for later declarations of structure variables.

A previously defined *identifier* (tag) can be used to refer to a structure type defined elsewhere. In this case, *struct-declaration-list* cannot be repeated 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 size of the fields. This is an incomplete definition of the type and the type tag. For this definition to be completed, a type definition must appear later in the same scope.

The *struct-declaration-list* specifies the types and names of the structure members. A *struct-declaration-list* argument contains one or more variable or bit-field declarations.

Each variable declared in *struct-declaration-list* is defined as a member of the structure type. Variable declarations within *struct-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 types except type **void**, an incomplete type, or a function type.

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.

Structures follow the same scoping as other identifiers. Structure identifiers must be distinct from other structure, union, and enumeration tags with the same visibility.

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

Nested structures can also be accessed as though they were declared at the file-scope level. For example, given this declaration:

```
struct a
}
    int x;
    struct b
    {
       int y;
    } var2;
} var1;
```

these declarations are both legal:

```
struct a var3;
struct b var4;
```

These examples illustrate structure declarations:

```
struct employee   /* Defines a structure variable named temp */
{
    char name[20];
    int id;
    long class;
} temp;
```

The `employee` 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 identifier.

```
struct employee student, faculty, staff;
```

This example 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 the previous example.

```
struct          /* Defines an anonymous struct and a
{                  structure variable named complex */
    float x, y;
} complex;
```

The `complex` structure has two members with **float** type, `x` and `y`. The structure type has no tag and is therefore unnamed or anonymous.

```
struct sample   /* Defines a structure named x */
{
    char c;
    float *pf;
    struct sample *next;
} 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`).

Anonymous structures can be useful when the tag named is not needed. This is the case when one declaration defines all structure instances. For example:

```
struct
{
    int x;
    int y;
} mystruct;
```

Embedded structures are often anonymous.

```
struct somestruct
{
    struct    /* Anonymous structure */
    {
        int x, y;
    } point;
    int type;
} w;
```

Microsoft Specific

The compiler allows an unsized or zero-sized array as the last member of a structure. This can be useful if the size of a constant array differs when used in various situations. The declaration of such a structure looks like this:

struct *identifier*
{
  set-of-declarations
  type array_name[ ];
};

Unsized arrays can appear only as the last member of a structure. Structures containing unsized array declarations can be nested within other structures as long as no further members are declared in any enclosing structures. Arrays of such structures are not allowed. The **sizeof** operator, when applied to a variable of this type or to the type itself, assumes 0 for the size of the array.

Structure declarations can also be specified without a declarator when they are members of another structure or union. The field names are promoted into the enclosing structure. For example, a nameless structure looks like this:

```
struct s
{
    float y;
    struct
    {
        int a, b, c;
    };
    char str[10];
} *p_s;
.
.
.
p_s->b = 100;  /* A reference to a field in the s structure */
```

See "Structure and Union Members" on page 119 for information about structure references.♦

Bit Fields

In addition to declarators for members of a structure or union, a structure declarator can also be a specified number of bits, called a "bit field." Its length is set off from the declarator for the field name by a colon. A bit field is intepreted as an integral type.

Syntax

struct-declarator :
  declarator
  type-specifier declarator $_{\text{opt}}$: *constant-expression*

The *constant-expression* specifies the width of the field in bits. The *type-specifier* for the *declarator* must be **unsigned int**, **signed int**, or **int**, and the *constant-expression* must be a nonnegative integer value. If the value is zero, the declaration has no *declarator.* Arrays of bit fields, pointers to bit fields, and functions returning bit fields are not allowed. The optional *declarator* names the bit field. Bit fields can only be declared as part of a structure. The address-of operator (**&**) cannot be applied to bit-field components.

Unnamed bit fields cannot be referenced, and their contents at run time are unpredictable. 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 struct-declaration list begins on an **int** boundary.

Bit fields must also be long enough to contain the bit pattern. For example, these two statements are not legal:

```
short a:17;        /* Illegal! */
int long y:33;   /* Illegal! */
```

This example defines a two-dimensional array of structures named `screen`.

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

The array contains 2,000 elements. Each element is an individual structure containing four bit-field members: `icon`, `color`, `underline`, and `blink`. The size of each structure is two bytes.

Microsoft Specific

Bit fields defined as **int** are treated as signed. A Microsoft extension to the ANSI C standard allows **char** and **long** types (both **signed** and **unsigned**) for bit fields. Unnamed bit fields with base type **long**, **short**, or **char** (**signed** or **unsigned**) force alignment to a boundary appropriate to the base type.

Bit fields are allocated within an integer from least-significant to most-significant bit. In the following code

```
struct mybitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test;
```

```
void main( void );
{
    test.a = 2;
    test.b = 31;
    test.c = 0;
}
```

the bits would be arranged as follows:

```
00000001 11110010
cccccccb bbbbaaaa
```

Since the 8086 family of processors stores the low byte of integer values before the high byte, the integer `0x01F2` above would be stored in physical memory as `0xF2` followed by `0x01`.♦

Storage and Alignment of Structures

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 spaces ("holes" or "padding") can appear between structure members in memory. The bit patterns appearing in such holes are unpredictable and can differ from structure to structure, or over time within a single structure.

Structure members are aligned to the minimum of their own size or the current packing size. For 16-bit targets, the default packing size is 2. This default corresponds to the /Zp2 command-line option.

32-Bit Specific

The default packing size is 4 for 32-bit targets.♦

Microsoft Specific

To conserve space, or to conform to existing data structures, you may want to store structures more or less compactly. The /Zp compiler option or the **pack** pragma controls how structure data is "packed" into memory. For more information on pragmas, see "Pragma Directives" on page 209.

Use the /Zp option to specify the same packing for all structures in a module. When you give the /Zp[*n*] option, where *n* is 1, 2, or 4, each structure member after the first is stored on *n*-byte boundaries, depending on the option you choose. If you use the /Zp option without an argument, structure members are packed on 1-byte boundaries.

On some processors, the /Zp option can result in slower program execution because of the time required to unpack structure members when they are accessed. For example, on an 8086 processor, this option can reduce efficiency if members with **int** or **long** type are packed in such a way that they begin on odd-byte boundaries.

To use the **pack** pragma to specify packing other than the packing specified on the command line for particular structures, give the **pack(** *n* **)** pragma, where *n* is 1, 2, or 4, before structures that you want to pack differently. To reinstate the packing given on the command line, specify the following with no arguments.

```
#pragma pack( )  /* Enables packing specified on command line */
```

See Chapter 12 in *Programming Techniques* for more information on packing.

For 16-bit targets, bit fields default to size **short**, which can cross a byte boundary but not a 16-bit or 32-bit boundary. If the size and location of a bit field would cause it to overflow the current integer, the field is moved to the beginning of the next available integer. If a bit field is declared as a **long**, it can hold up to 32 bits. In either case, an individual field cannot cross a 16- or 32-bit boundary.♦

32-Bit Specific

Bit fields default to size **long** for the 32-bit compiler.♦

Union Declarations

A "union declaration" specifies a set of variable values and, optionally, a tag naming the union. The variable values are called "members" of the union and can have different types. Unions are similar to "variant records" in other languages.

Syntax

struct-or-union-specifier :
  struct-or-union identifier $_{\text{opt}}$**{** *struct-declaration-list* **}**
  struct-or-union identifier

struct-or-union :
  struct
  union

struct-declaration-list :
  struct-declaration
  struct-declaration-list struct-declaration

The union content is defined to be

struct-declaration :
  specifier-qualifier-list struct-declarator-list **;**

specifier-qualifier-list :
  type-specifier specifier-qualifier-list $_{\text{opt}}$
  type-qualifier specifier-qualifier-list $_{\text{opt}}$

struct-declarator-list :
  struct-declarator
  struct-declarator-list **,** *struct-declarator*

A variable with **union** type stores one of the values defined by that type. The same rules govern structure and union declarations. Unions can also have bit fields.

Members of unions cannot have an incomplete type, type **void**, or function type. Therefore members cannot be an instance of the union but can be pointers to the union type being declared.

A union type declaration is a template only. Memory is not reserved until the variable is declared.

Note If a union of two types is declared and one value is stored, but the union is accessed with the other type, the results are unreliable. For example, a union of **float** and **int** is declared. A **float** value is stored, but the program later accesses the value as an **int**. In such a situation, the value would depend on the internal storage of **float** values. The integer value would not be reliable.

The following are examples of unions:

```
union sign   /* A definition and a declaration */
{
    int svar;
    unsigned uvar;
} number;
```

This example defines a union variable 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`.

```
union                /* Defines a two-dimensional */
{                    /*  array named screen */
    struct
    {
       unsigned int icon : 8;
       unsigned color : 4;
    } window1;
    int screenval;
} screen[25][80];
```

The `screen` array contains 2,000 elements. Each element of the array 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`.

Microsoft Specific Nested unions can be declared anonymously when they are members of another structure or union. This is an example of a nameless union:

```
struct str
{
    int a, b;
    union            / * Unnamed union */
    {
       char c[4];
       long l;
       float f;
    };
    char c_array[10];
} my_str;
.
.
.
my_str.l == 0L;  /* A reference to a field in the my_str union */
```

Unions are often nested within a structure that includes a field giving the type of data contained in the union at any particular time. This is an example of a declaration for such a union:

```
struct x
{
    int type_tag;
    union
    {
       int x;
       float y;
    }
}
```

See "Structure and Union Members" on page 119 for information about referencing unions.♦

Storage of Unions

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 can 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. For example:

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

The members of the `jack` 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 with the union, its type is unnamed or "anonymous."

Array Declarations

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.

Syntax

declaration :
  declaration-specifiers init-declarator-list $_{\text{opt}}$ **;**

init-declarator-list **:**
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator **:**
  declarator
  declarator = *initializer*

declarator :
  pointer $_{\text{opt}}$ *direct-declarator*

direct-declarator :
  direct-declarator **[** *constant-expression* $_{\text{opt}}$ **]**

Because *constant-expression* is optional, the syntax has two forms:

- The first form defines an array variable. The *constant-expression* argument within the brackets specifies the number of elements in the array. The *constant-expression*, if present, must have integral type, and a value larger than zero. Each element has the type given by *type-specifier*, which can be any type except **void**. An array element cannot be a function type.
- The second form declares a variable that has been defined elsewhere. It omits the *constant-expression* argument in brackets, but not the brackets. You can use this form only if you previously have initialized the array, declared it as a parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

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

Type qualifiers can appear in the declaration of an object of array type, but the qualifiers apply to the elements rather than the array itself.

You can declare an array of arrays (a "multidimensional" array) by following the array declarator with a list of bracketed constant expressions in this form:

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. You can omit the first constant expression if you have initialized the array, declared it as a 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 "Interpreting More Complex Declarators" on page 88.

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, as described in "Postfix Operators" on page 116. These examples illustrate array declarations:

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

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

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

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

```
extern char *name[];
```

This statement declares the type and name of an array of pointers to **char**. The actual definition of `name` occurs elsewhere.

Microsoft Specific

The type of integer required to hold the maximum size of an array is the size of **size_t**. The **size_t** type definition for 16-bit targets is an **unsigned short**, with the range 0x0000 to 0xFFFF hexadecimal. Huge arrays can exceed this limit if they contain more than 65,535 elements or the size of the element multiplied by the number of elements is greater than 65K. Arithmetic operations on arrays specified with the __**huge** keyword should therefore cast **size_t** and the results of an arithmetic operation on pointers to **unsigned long**. ♦

32-Bit Specific

For 32-bit targets, **size_t** is **unsigned long** and the __**huge** keyword is not required. ♦

Storage of Arrays

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.

Pointer Declarations

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.

Syntax

declarator **:**
  pointer $_{\text{opt}}$ *direct-declarator*

direct-declarator :
  identifier
  (*declarator* **)**
  direct-declarator **[** *constant-expression* $_{\text{opt}}$ **]**
  direct-declarator **(** *parameter-type-list* **)**
  direct-declarator **(** *identifier-list* $_{\text{opt}}$ **)**

pointer :
  * *type-qualifier-list* $_{\text{opt}}$
  * *type-qualifier-list* $_{\text{opt}}$ *pointer*

type-qualifier-list :
  type-qualifier
  type-qualifier-list type-qualifier

The *type-specifier* gives the type of the object, which can be any basic, structure, or union type. Pointer variables can also point to functions, arrays, and other pointers. (For information on declaring and interpreting more complex pointer types, refer to "Interpreting More Complex Declarators" on page 88.)

By making the *type-specifier* **void**, you can delay specification of the type to which the pointer refers. Such an item is referred to as a "pointer to **void**" and is written as `void *`. A variable declared as a pointer to **void** can be used to point to an object of any type. However, to perform most 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. (Variables of type **char** * and type **void** * are assignment-compatible without a type cast.) Such conversion can be accomplished with a type cast (see "Type-Cast Conversions" on page 147).

The *type-qualifier* can be either **const** or **volatile**, or both. These specify, respectively, that the pointer cannot be modified by the program itself (**const**), or that the pointer can legitimately be modified by some process beyond the control of the program (**volatile**). (See "Type Qualifiers" on page 52 for more information on **const** and **volatile**.)

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

You can declare a pointer to a structure, union, or enumeration type before you define the structure, union, or enumeration type. You declare the pointer by using the structure or union tag as shown in the examples 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.

The following examples illustrate pointer declarations.

```
char *message; /* Declares a pointer variable named message */
```

The `message` pointer points to a variable with **char** type.

```
int *pointers[10];  /* Declares an array of pointers */
```

The `pointers` array has 10 elements; each element is a pointer to a variable with **int** type.

```
int (*pointer)[10]; /* Declares a pointer to an array of 10 elements */
```

The pointer variable points to an array with 10 elements. Each element in this array has **int** type.

```
int const *x;      /* Declares a pointer variable, x,
                      to a constant value */
```

The pointer `x` can be modified to point to a different **int** value, but the value to which it points cannot be modified.

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

The variable `y` in these declarations is declared as a constant pointer to an **int** value. The value it points to can 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 cannot 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 running concurrently with the program. The declaration of `w` specifies that the program cannot change the value pointed to and that the program cannot modify the pointer.

```
struct list *next, *previous; /* Uses the tag for list */
```

This example declares two pointer variables, `next` and `previous`, that point to the structure type `list`. This declaration can appear before the definition of the `list` structure type (see the next example), as long as the `list` type definition has the same visibility as the declaration.

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

The variable `line` has the structure type named `list`. The `list` structure type has 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.

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

The variable `record` has 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.

Storage of Addresses

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. Therefore, **sizeof(char *)** is not necessarily equal to **sizeof(int *)**.

Microsoft Specific

For the Microsoft C compiler, **sizeof(char *)** is equal to **sizeof(int *)**. You can also use the special keywords **__near**, **__far**, **__huge**, and **__based** for 16-bit targets to modify and reference the size of a pointer. Declarations using special keywords are described in "Special Keywords in Declarations" on page 55.♦

Based Pointers (Microsoft Specific)

For the 16-bit compiler, a based pointer operates as a 16-bit offset from a base you specify. For the 32-bit compiler, a based pointer is a 32-bit offset from a 32-bit pointer base. In this respect, based addressing differs from near, far, or huge addressing because you are responsible for naming the base.

Based addressing is useful for exercising control over segments where objects are allocated and for referencing far objects with 16-bit addresses, thereby decreasing the size of the executable file and increasing execution speed.

The **__based** keyword specifies that a pointer is a 16-bit value interpreted as an offset from a specified base, or that a data object resides in the segment given by a specified base. In general, the form for specifying a based pointer is

type **__based(** *base* **)** *declarator*

The following code shows some examples of based pointer declarations:

```
typedef struct tree BTree;

struct tree    /* Binary tree */
{
    char *szSymbolName;
    BTree *btLeft;
    BTree *btRight;
};

/* Pointer to data that resides in segment SYM_DATA: */

BTree __based( __segname( "SYM_DATA" ) ) *btSymTable1;

/* Pointer to data that resides in same segment
   as the pointer btSymTable3: */

BTree __based( (__segment) __self ) *btSymTable3;
```

Based pointers can address any location in memory but are only two bytes in size for 16-bit targets (4 bytes for 32-bit targets), because they contain only the offset portion of an address. The segment portion of the address is stored separately and is combined with the offset when needed. Multiple based pointers can share the same segment value, so they require less memory than far pointers and allow the compiler to generate better code.

You also can use the **__based** keyword to specify the segment in which data resides.

A based pointer can be based on a fixed segment, a variable segment, self, void, or a pointer. The following sections explain these topics. See Chapter 4 in *Programming Techniques* for additional information on using based addressing.

The __segname and __segment Keywords (Microsoft Specific)

The built-in function **__segname** accepts a quoted string and returns a value of type **__segment**. This built-in function can be used to initialize variables of type **__segment** or in declarations of based identifiers or pointers. A **__segname** declaration is not an l-value, and its address cannot be taken. The primary use for **__segname** is in initializing identifiers of type **__segment** or in declarations of pointers or identifiers of based type. It enables specification of a segment name instead of a segment value when declaring a based type.

The **__segment** keyword is a type that contains a segment value. Variables of such type can contain a segment value. A variable or value of that type is used either at the point of declaration of a based type or at the point of a based dereference to specify the segment in which the based identifier resides.

An identifier of type **__segment** can be initialized with the following:

- An expression that evaluates to an integral constant value.
- The result of the built-in function **__segname**. This allows specification of a segment by name, causing the linker to insert segment fixups in the executable file, to be resolved at load time. This kind of initialization can be performed only on static and external identifiers, not identifiers with block scope.
- Another expression of type **__segment**. For example:

```
main()
{
    __segment sgCustomerData = 0x7000;
    __segment sgCurrent = sgCustomerData;
}
```

- Another expression explicitly cast to type **__segment**.

If specified at the point of declaration, the segment value is implicit and need not be respecified at the point of dereference (based pointers). Otherwise, the segment must be explicitly specified.

32-Bit Specific

The **__segment** type and the built-in function **__segname** are not supported by the 32-bit compiler.♦

Pointers Based on a Constant (Microsoft Specific)

Pointers based on a constant are restricted to accessing one segment of memory. By making assignments to the based pointer, you change only the offset portion of the address. To specify a pointer based on a fixed segment, you can use the **__segname** built-in function or the **__segment** keywords to supply the segment name.

Pointers based on a named segment are specified as

__segname(*string-literal* **)**

The *string-literal* can be the name of one of four predefined segments (_CODE, _CONST, _DATA, or _STACK), or it can be the name of a new segment you define. A based pointer declared this way can address locations in only the specified segment.

Pointers based on the segment of a variable are specified as

(__segment)&*var*

A based pointer declared this way uses the segment of the address of *var* as its base. It can address locations only in the same segment as *var*.

In the following example, `sgConst` is explicitly declared as a constant. The pointer `bp1`, therefore, is based on a constant.

```
const __segment sgConst = 0x3000;
char __based( sgConst ) *bp1;
```

In the following example, the **__segname** function returns a constant value of type **__segment**. The pointer `bp3`, therefore, is based on a constant.

```
char __based( __segname("INFO_STRINGS") ) *bp3;
```

Pointers Based on a Segment Variable (Microsoft Specific)

Pointers based on a nonfixed segment have access to locations in any segment simply by changing the value of the base. Changing a single segment value causes all pointers based on that segment to address new locations. You also can make assignments to the based pointers themselves to change their offset values.

Pointers based on a segment variable require that a variable of type **__segment** be declared. This variable determines what segment the based pointer refers to. The segment variable can be changed at run time.

type **__based(__segment) *** *ptr*

A based pointer declared this way uses the segment portion of *ptr* as its base. If *ptr* is a near pointer, the declared pointer uses the DS register as its base. If *ptr* is a far pointer, the declared pointer uses the segment of the *ptr* as its base. Changing the segment value of *ptr* causes the based pointer to address a new location.

The base for pointers based on a segment variable is specified as

__segment *segvar*

A based pointer declared this way uses *segvar* as its base. Assigning a new value to *segvar* causes the based pointer to address a different location. This type of based pointer can be used for dynamic allocation of based identifiers.

The form of *base* of pointers based on another pointer is

*type * ptr*

A based pointer declared this way acts as an offset from *ptr*. Assigning a new value to *ptr* causes the based pointer to address a different location.

Pointers Based on Pointers (Microsoft Specific)

The "based on pointer" variant of based addressing enables specification of a pointer as a base. The based pointer, then, is an offset into the segment starting at the beginning of the pointer on which it is based.

One use for pointers based on pointers is for persistent identifiers that contain pointers. A linked list that consists of pointers based on a pointer can be saved to disk, then reloaded to another place in memory, with the pointers remaining valid.

The following example shows a pointer based on a pointer.

```
void *vpBuffer;

struct llist_t
{
    void __based( vpBuffer ) *vpData;
    llist_t __based( vpBuffer ) *llNext;
};
```

The pointer `vpBuffer` is assigned the address of memory allocated at some later point in the program. The linked list is relocated relative to the value of `vpBuffer`.

32-Bit Specific

Pointers based on pointers are the only form of the **__based** keyword valid in 32-bit compilations. In such compilations, they are 32-bit displacements from a 32-bit base.♦

Pointers Based on void (Microsoft Specific)

Pointers based on **void** defer actual address calculation until the pointer is dereferenced. A pointer based on **void** acts as an offset into any segment. The form of a pointer based on **void** is *type* **__based(void)** * *ptr*

A pointer based on **void** has no implied segment as its base. The segment specified at the point of dereference can be a constant or a segment variable. The segment and offset are combined using the base operator (**:>**) to form an address that can be dereferenced using the indirection operator (*).

```
struct BiosEquipList  /* Structure for the BIOS Equipment List
{                      * that starts at 000:0410 (hex)
    .                  */
    . /* structure fields */
    .
};

/* Declare ROM data as const and supply the offset, hex 410: */

const BiosEquipList __based( void ) *bpelROM = 0x410;

int main()
{
    BiosEquipList elLocal;  /* Local copy of equipment list */

elLocal = *(0x000 :> bpelROM); /* Segment and offset combined */
```

This example shows how to declare and dereference a pointer based on **void**.

Pointers Based on the __self keyword (Microsoft Specific)

Pointers based on self can access data anywhere in the segment in which the pointer resides. They are declared using the **__self** keyword cast to the **__segment** type as the base. You can only base on **(__segment)__self**; not on **__self** only. Basing a pointer on **(__segment)__self** can improve program performance by requiring that the segment register be the same for addressing both the pointer and the data it addresses. Functions cannot return pointers based on self.

The **__segment** keyword must be used to cast to a segment value, as in the example below:

```
typedef struct tree TREE;

struct tree
{
    int name;
    TREE __based( (__segment)__self ) *left;
    TREE __based( (__segment)__self ) *right;
};

TREE __based( __segname( "MYSEGMENT" ) ) t1;
```

The example above declares a structure called `tree` and then declares `t1` to be such a structure. The pointers within the structure are self-based, meaning that they point within the segment in which the `tree` structure is located.

Pointers based on (**__segment)__self** are particularly useful for optimizing access in self-referencing data structures such as linked-lists and trees.

Any based declarations that are based on (**__segment)__self** must apply to pointers only. Ordinary data identifiers cannot be self-based.

32-Bit Specific

Pointers based on **__self** are not available for 32-bit targets.♦

Function Declarations

A "function declaration" 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 information about the function known to the compiler. This information enables the compiler to check the types of the actual arguments passed in calls to the function. Functions are declared with declarators:

Syntax

declarator :
  pointer $_{\text{opt}}$ *direct-declarator*

direct-declarator : /* A function declarator */
  direct-declarator (*parameter-type-list*) /* New-style declarator */
  direct-declarator (*identifier-list* $_{\text{opt}}$) /* Old-style declarator */

parameter-type-list :
  parameter-list
  parameter-list , **...**

parameter-list :
  parameter-declaration
  parameter-list , *parameter-declaration*

parameter-declaration :
  declaration-specifiers declarator /* Named declarator */
  declaration-specifiers abstract-declarator $_{\text{opt}}$ /* Anonymous declarator */

identifier-list : /* For old-style declarator */
  identifier
  identifier-list , *identifier*

declaration-specifiers :
  storage-class-specifier declaration-specifiers opt
  type-specifier declaration-specifiers opt
  type-qualifier declaration-specifiers opt

abstract-declarator : /* Used with anonymous declarators */
  pointer
  pointer opt *direct-abstract-declarator*

direct-abstract-declarator :
  (*abstract-declarator*)
  direct-abstract-declarator opt [*constant-expression* opt]
  direct-abstract-declarator opt (*parameter-type-list* opt)

If specified, *storage-class-specifier* can be either **extern** or **static**. Storage-class specifiers are discussed in "Storage Classes" on page 43. The *type-specifier* gives the function's return type, and *declarator* names the function. If you omit *type-specifier* from a function declaration, the function is assumed to return a value of type **int**. The *parameter-type-list* is described below in "Parameters."

Other declarators can appear in the same function declaration. These can be other functions returning values of the same type as the function, or declarations of any variables whose type is the same as the function's return type. Each such declaration must be separated from its predecessors and successors by a comma.

A function prototype gives information about the parameters, allowing the compiler to perform type checking and to convert arguments to the type expected by the parameter. The function definition defines the body of the function.

Parameters

"Parameters" (sometimes called "formal parameters") describe the actual arguments that can be passed to a function. In a parameter-type list, the parameter declarations establish the number and types of the actual arguments. They can also include identifiers of the formal parameters.

Note 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.

Although the parameters may be omitted from a function declaration in the optional *identifier-list* form of the syntax, their inclusion is recommended. The extent of the information in the declaration influences the argument checking done on function calls that appear before the compiler has processed the function definition.

If a function has no parameters, the parentheses should contain the keyword **void** to specify that no arguments are passed to the function.

The only explicit *storage-class-specifier* permitted in parameters is **register**. If **register** is not specified, the storage class is **auto**. The **register** specifier is ignored unless the function declarator has a function definition. If the parentheses contain only the **register** keyword, the parameter is considered to represent an unnamed **int** for which **register** storage is being requested.

The *declarator* for a pointer, array, or function can be formed by combining a type specifier, plus the appropriate type qualifier, with an identifier. Alternatively, an *abstract-declarator* (that is, a declarator without a specified identifier) can be used. Complete declarators (`int a`) and abstract declarators (`int`) are permitted in the same prototype. For example:

```
int func( int a, int );  /* Accepted */
```

In a parameter declaration, a single typedef name inside parentheses is assumed to be an abstract declarator specifying a function with a single parameter, not as redundant parentheses around an identifier for the declarator. See page 88 for information about abstract declarators.

One other special construction permitted as a parameter is **void ***, representing a pointer to an identifier of unspecified type. Thus, in a call, the pointer can be used to pass any type of identifier after you convert 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 the identifier it addresses, the pointer must be explicitly converted. "Pointer Declarations" on page 76 provides further information on **void ***.

The list of parameters can be empty, full, or partial. 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 notation." No information about the number or type of the parameters after the comma is supplied. See "Calls with a Variable Number of Arguments" on page 187 for information about functions with variable numbers of arguments.

Microsoft Specific

To maintain compatibility with previous versions of the Microsoft C compiler, the version 7.0 compiler accepts a comma without trailing periods at the end of a declarator list to indicate a variable number of arguments. (Trailing periods without a comma are not allowed.) However, this is a Microsoft extension to the ANSI C standard. New code should use the comma followed by three periods. ANSI also requires at least one argument before the ellipsis. The STDARG.H file enforces this restriction for accessing arguments. ♦

Return Types

Functions can return values of any type except arrays and functions. Therefore, the *type-specifier* argument of a function declaration can specify any basic, structure, or union type. You can modify the function identifier with one or more asterisks (*) to declare a pointer return type. Functions declared as **float** always return a value of type **float**. In earlier versions of the Microsoft C compiler, the return value of a function using the obsolete form of a function declaration and returning type **float** was converted to type **double**. The Microsoft C version 7.0 compiler conforms to the ANSI standard by not changing the return type.

Although functions cannot return arrays and functions, they can return pointers to arrays and functions. You can 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 "complex declarator." Rules for forming and interpreting complex declarators are discussed in "Interpreting More Complex Declarators" on page 88.

The following examples illustrate return types in function declarations:

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

The `draw` function returns a **void** type (returns no value). The **void** keyword also replaces the list of parameters so no arguments are allowed for this function.

```
double ( *sum(double, double) )[3];
```

In this example, `sum` is declared as a function returning a pointer to an array of three **double** values. The `sum` function takes two **double** values as arguments.

```
int ( *select(void) )( int number );
```

The function named `select` takes no arguments and returns 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.

```
int prt( void * );
```

The function `prt` takes a pointer argument of any type and returns an **int** value. A pointer to any type could be passed as an argument to `prt` without producing a type-mismatch warning.

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

This array, named `rainbow`, contains an unspecified number of constant pointers to functions. Each of these takes 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.

Abstract Declarators

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 "Interpreting More Complex Declarators" on page 88 for additional information and examples of complex declarators. Generally **typedef** can be used to simplify declarators. See "Typedef Declarations" on page 101.

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.

These examples illustrate abstract declarators:

```
int *         /* The type name for a pointer to type int:   */

int *[3]      /* An array of three pointers to int          */

int (*) [5]   /* A pointer to an array of five int          */

int *()       /* A function with no parameter specification */
              /* returning a pointer to int                 */

/* A pointer to a function taking no arguments and
 * returning an int
 */

int (*) ( void )

/* An array of an unspecified number of constant pointers to
 * functions each with one parameter that has type unsigned int
 * and an unspecified number of other parameters returning an int
 */

int (*const []) ( unsigned int, ... )
```

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).

3.6 Interpreting More 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. Generally **typedef** may be used to simplify declarations. See "Typedef Declarations" on page 101.

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. By using parentheses you can override the default association order and force a particular interpretation. Never use parentheses, however, around an identifier name by itself. This could be misinterpreted as a parameter list.

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 directly 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.
4. Apply the type specifier.

```
char *( *(*var)() )[10];
 ^   ^  ^ ^ ^   ^    ^
 7   6  4 2 1   3    5
```

In this example, the steps are numbered 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.

The following examples illustrate other complex declarations and show how parentheses can affect the meaning of a declaration.

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

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.

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

In this declaration for `var`, 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.

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

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

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

This example is similar to the previous one. 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.

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

The elements of an array cannot be functions, but this declaration 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 );
```

The following statement declares an array of pointers.

```
unsigned int *(* const *name[5][10] ) ( void );
```

The `name` array has 50 elements organized in a multidimensional array. The elements are pointers to a pointer that is a constant. This constant pointer points to a function that has no parameters and returns a pointer to an unsigned type.

This next example is a function returning a pointer to an array of three double values.

```
double ( *var( double (*)[3] ) )[3];
```

In this declaration, a function returns 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 page 88.

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

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

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

This example 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. For examples of how to use **typedef** to avoid complex declarations, see "Typedef Declarations" on page 101.

3.7 Initialization

An "initializer" is a value or a sequence of values to be assigned to the variable being declared. 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 following sections describe how to initialize variables of scalar, aggregate, and string types. "Scalar types" include all the arithmetic types, plus pointers. "Aggregate types" include arrays, structures, and unions.

Scalar Initialization

When initializing scalar types, the value of the assignment-expression is assigned to the variable. The conversion rules for assignment apply. (See"Type Conversions" on page 141 for information on conversion rules.)

Syntax

declaration :
  declaration-specifiers init-declarator-list $_{\text{opt}}$ **;**

declaration-specifiers :
  storage-class-specifier declaration-specifiers $_{\text{opt}}$
  type-specifier declaration-specifiers $_{\text{opt}}$
  type-qualifier declaration-specifiers $_{\text{opt}}$

init-declarator-list :
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator :
  declarator
  declarator **=** *initializer*   /* For scalar initialization */

initializer :
  assignment-expression

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

- Variables declared at the file-scope level can be initialized. If you do not explicitly initialize a variable at the external level, it is initialized to 0 by default.
- A constant expression can be used to initialize any global 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 global **static** variable, it is initialized to 0 by default, and every member that has pointer type is assigned a null pointer.
- Variables declared with the **auto** or **register** storage-class specifier 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. For automatic and register values, the initializer is not restricted to being a constant; it can be any expression involving previously defined values, even function calls.
- The initial values for external variable declarations and for all **static** variables, whether external or internal, must be constant expressions. (Constant expressions are described on page 108.) Since the address of any externally declared or static variable is constant, it can be used to initialize an internally declared **static** pointer variable. However, the address of an **auto** variable cannot be used as a static initializer because it may be different for each execution of the block. You can use either constant or variable values to initialize **auto** and **register** variables.
- If the declaration of an identifier has block scope, and the identifier has external linkage, the declaration cannot have an initialization.

The following examples illustrate initializations:

```
int x = 10;
```

The integer variable `x` is initialized to the constant expression `10`.

```
register int *px = 0;
```

The pointer `px` is initialized to 0, producing a "null" pointer.

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

This example uses a constant expression `(3 * 1024)` to initialize `c` to a constant value that cannot be modified because of the **const** keyword.

```
int *b = &x;
```

This statement initializes the pointer `b` with the address of another variable, `x`.

```
int *const a = &z;
```

The pointer `a` is initialized with the address of a variable named `z`. However, since it is specified to be a **const**, the variable `a` can only be initialized, never modified. It always points to the same location.

```
int GLOBAL ;

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

The global variable `GLOBAL` is declared 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 it 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. Similarly, `*rp` can be initialized because `rp` is a local variable and can have a nonconstant initializer. Each time the block is entered, `LOCAL` has a new address, which is then assigned to `rp`.

Initializing Aggregate Types

An "aggregate" type is a structure, union, or array type. If an aggregate type contains members of aggregate types, the initialization rules apply recursively.

Syntax

initializer :
  { *initializer-list* } /* For aggregate initialization */
  { *initializer-list* , }

initializer-list :
  initializer
  initializer-list , *initializer*

The *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, initializer lists can be nested. This form is useful for initializing aggregate members of an aggregate type, as shown in the examples in this section. However, if the initializer for an automatic identifier is a single expression, it need not be a constant expression; it merely needs to have appropriate type for assignment to the identifier.

For each initializer list, the values of the constant expressions are assigned, in order, to the corresponding members of the aggregate variable.

If *initializer-list* has fewer values than an aggregate type, the remaining members or elements of the aggregate type are initialized to 0 for external and static variables. The initial value of an automatic identifier not explicitly initialized is undefined. If *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.

A structure's initializer is either an expression of the same type, or a list of initializers for its members enclosed in curly braces ({ }). Unnamed bit-field members are not initialized.

When a union is initialized, *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 array has unknown size, the number of initializers determines the size of the array, and its type becomes complete. There is no way to specify repetition of an initializer in C, or to initialize an element in the middle of an array without providing all preceding values as well. If you need this operation in your program, write the routine in assembly language.

Note that the number of initializers can set the size of the array:

```
int x[ ] = { 0, 1, 2 }
```

If you specify the size and give the wrong number of initializers, however, the compiler generates an error.

The maximum size for an array is defined by **size_t**, an **unsigned short** on 16-bit computers, and has the range 0x0000 to 0xFFFF hexadecimal. Huge arrays can exceed this limit if they contain more than 65,535 elements or if the size of the element multiplied by the number of elements exceeds 65K. Arithmetic operations on huge arrays should therefore cast **size_t** and the results of arithmetic operations to **unsigned long**.

32-Bit Specific

On 32-bit computers, **sizeof** is **unsigned long**.♦

This example shows initializers for an array.

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

This statement declares `P` as a four-by-three 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 and would help to clarify the example above.

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 } },   /* Row 1 */
    { { 10,11,12 }, { 13,14,15 }, { 16,17,18 } }    /* Row 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:

```
triplet nlist[2][3] =  /* THIS CAUSES AN ERROR */
{
     {  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.

In this next example, the three **int** members of `x` are initialized to 1, 2, and 3, respectively.

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

In the `list` structure above, 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.

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

The union variable `y`, in this example, 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 the value 0 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 the value 0.

Initializing Strings

You can initialize an array of characters (or wide characters) with a string literal (or wide string literal). For example:

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

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

An identifier list can only be as long as the number of identifiers to be initialized. If you specify an array size that is shorter than the string, the extra characters are 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. Note that this creates an unterminated string (that is, one without a 0 value to mark its end) and generates a diagnostic message indicating this condition.

The declaration

```
char s[] = "abc", t[3] = "abc";
```

is identical to

```
char s[]  = {'a', 'b', 'c', '\0'},
     t[3] = {'a', 'b', 'c' };
```

If the string is shorter than the specified array size, the remaining elements of the array are initialized to 0. String literals can be up to 4K in length.

3.8 Storage of Basic Types

Table 3.2 summarizes the storage associated with each basic type.

Table 3.2 Sizes of Fundamental Types

Type	16-Bit Targets	32-Bit Targets
char, **unsigned char**, **signed char**	1 byte	1 byte
short, **short int**, **signed short**, **unsigned short**	2 bytes	2 bytes
int, **unsigned int**, **signed int**	2 bytes	4 bytes
long, **unsigned long**, **signed long**	4 bytes	4 bytes
float	4 bytes	4 bytes
double	8 bytes	8 bytes
long double	10 bytes	10 bytes

The C data types fall into general categories. The "integral types" include **char**, **int**, **short**, **long**, **signed**, **unsigned**, and **enum**. The "floating types" include **float**, **double**, and **long double**. The "arithmetic types" include all floating and integral types.

Type char

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.

Microsoft Specific Character values of type **unsigned char** have a range from 0 to 0xFFh. A **signed char** has range 0x80h to 0x7Fh. These ranges translate to 0 to 255 decimal, and –128 to +127, decimal, respectively. The /J command-line option changes the default from **signed** to **unsigned**.♦

Type int

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 computer, 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 target environment. The **int** types all represent signed values unless specified otherwise.

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

Microsoft Specific Signed integers are represented in two's-complement form. The most-significant bit holds the sign: 1 for negative, 0 for positive and zero. The range of values is given in Table 1.3, which is taken from the LIMITS.H header file.♦

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. To make programs more portable, you can use expressions with the **sizeof** operator (as discussed in "The sizeof Operator" on page 125) instead of hard-coded data sizes.

Type float

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers) format. Single-precision 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**.

Microsoft Specific The **float** type contains 32 bits: 1 for the sign, 8 for the exponent, and 23 for the mantissa. Its range is +/–3.4E38 with at least 7 digits of precision.♦

Type double

Double precision 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**.

Microsoft Specific The **double** type contains 64 bits: 1 for sign, 11 for the exponent, and 52 for the mantissa. Its range is +/–1.7E308 with at least 15 digits of precision.♦

Type long double

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 "Type Conversions" on page 141) 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.

The value 65,000 in decimal notation is considered a signed constant. It is given the **long** type because 65,000 does not fit into a **short**. A value such as 65,000 can only be represented as an **unsigned short** by type-casting the value to **unsigned short** type, by giving the value in octal or hexadecimal notation, or by specifying it as 65000U. 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.

Microsoft Specific The **long double** contains 80 bits: 1 for sign, 15 for exponent, and 64 for mantissa. Its range is +/–1.2E4932 with at least 17 digits of precision.♦

3.9 Incomplete Types

An incomplete type is a type that describes an identifier but lacks information needed to determine the size of the identifier. An "incomplete type" can be

- A structure type whose members you have not yet specified
- A union type whose members you have not yet specified
- An array type whose dimension you have not yet specified

The **void** type is an incomplete type that cannot be completed. To complete an incomplete type, specify the missing information. The following examples show how to create and complete the incomplete types.

- To create an incomplete structure type, declare a structure type without specifying its members. In this example, the `ps` pointer points to an incomplete structure type called `student`.

  ```
  struct student *ps;
  ```

- To complete an incomplete structure type, declare the same structure type later in the same scope with its members specified, as in

  ```
  struct student
  {
      int num;
  }                   /* student structure now completed */
  ```

- To create an incomplete array type, declare an array type without specifying its repetition count. For example:

  ```
  char a[];  /* a has incomplete type */
  ```

- To complete an incomplete array type, declare the same name later in the same scope with its repetition count specified, as in

  ```
  char a[25]; /* a now has complete type */
  ```

3.10 Typedef Declarations

A typedef declaration is a declaration with **typedef** as the storage class. The declarator becomes a new 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. Typedef names allow you to encapsulate implementation details that may change.

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.

Syntax

declaration :
  declaration-specifiers init-declarator-list $_{opt}$ **;**

declaration-specifiers :
  storage-class-specifier declaration-specifiers $_{opt}$
  type-specifier declaration-specifiers $_{opt}$
  type-qualifier declaration-specifiers $_{opt}$

storage-class-specifier :
  typedef

type-specifier :
  void
  char
  short
  int
  long
  float
  double
  signed
  unsigned
  struct-or-union-specifier
  enum-specifier
  typedef-name

typedef-name :
  identifier

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** and **volatile**, and the special keywords described in "Special Keywords in Declarations" on page 55.)

Typedef names share the name space with ordinary identifiers (see "Name Spaces" on page 39 for more information). Therefore, a program can have a typedef name and a local scope identifier by the same name. For example:

```
typedef char FlagType;

int main()
{
}

int myproc( int )
{
    int FlagType;
}
```

When declaring a local-scope identifier by the same name as a typedef, or when declaring a member of a structure or union in the same scope or in an inner scope, the type specifier must be specified. This example illustrates this constraint:

```
typedef char FlagType;
const FlagType x;
```

To reuse the `FlagType` name for an identifier, a structure member, or a union member, the type must be provided:

```
const int FlagType;  /* Type specifier required */
```

It is not sufficient to say

```
const FlagType;      /* Incomplete specification */
```

because the `FlagType` is taken to be part of the type, not an identifier that is being redeclared. This declaration is taken to be an illegal declaration like

```
int;  /* Illegal declaration */
```

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.

Typedef names can be used to improve code readability. All three of the following declarations of `signal` specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv( int ), (*pfv)( int );  /* typedef declarations */

void ( *signal( int, void (*) (int)) ) ( int );
fv *signal( int, fv * );   /* Uses typedef type */
pfv signal( int, pfv );    /* Uses typedef type */
```

The following examples illustrate **typedef** declarations:

```
typedef int WHOLE; /* Declares WHOLE to be a synonym for int */
```

Note that `WHOLE` could now be used in a variable declaration such as `WHOLE i;` or `const WHOLE i;`. However, the declaration `long WHOLE i;` would be illegal.

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

This statement 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. You must use the **struct** keyword with the tag, and you cannot use the **struct** keyword with the **typedef** name.

```
typedef GROUP *PG; /* Uses the previous typedef name
                      to declare a pointer            */
```

The type `PG` is declared as a pointer to the `GROUP` type, which in turn is defined as a structure type.

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

This example 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 );
```

Expressions and Assignments

Chapter 4

This chapter describes how to form expressions and to assign values in the C language. Constants, identifiers, strings, and function calls are all operands that are manipulated in expressions. The C language has all the usual language operators. This chapter covers those operators plus operators such as the conditional operator and base operator that are unique to C or Microsoft C. The topics discussed in this chapter include

- L-value and r-value expressions
- Constant expressions
- Side effects
- Sequence points
- Expression evaluation
- Operators and operator precedence
- Type conversions
- Type casts

4.1 Operands and Expressions

An "operand" is an entity on which an operator acts. An "expression" is a sequence of operators and operands that performs any combination of these actions:

- Computes a value
- Designates an object or function
- Generates side effects

Operands in C include constants, identifiers, strings, function calls, subscript expressions, member-selection expressions, and complex expressions formed by combining operands with operators or by enclosing operands in parentheses.

These operands, often called "primary expressions," are discussed in the next section.

Primary Expressions

The operands in expressions are called "primary expressions."

Syntax

primary-expression :
  identifier
  constant
  string-literal
  (*expression*)

expression :
  assignment-expression
  expression , *assignment-expression*

Identifiers

Identifiers can have integral, **float**, **enum**, **struct**, **union**, array, pointer, or function type. An identifier is a primary expression provided it has been declared as designating an object (in which case it is an l-value) or as a function (in which case it is a function designator). See the next section for a definition of l-value.

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 and therefore is not a modifiable l-value.

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. Thus, function identifiers also cannot be l-values in assignment operations. See page 5 for more information about identifiers.

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 on the way the value is specified. See "Constants" on page 9 for more information.

String Literals

A "string literal" is a character, wide character, or sequence of adjacent characters enclosed in double quotation marks. Since they are not variables, neither string literals nor any of their elements can be the left-hand operand in an assignment operation. The type of a string literal is an array of **char** (or an array of **wchar_t** for wide string literals. Arrays in expressions are converted to pointers. See "String Literals" on page 20 for more information about strings.

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 evaluated first and it becomes 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. For example, neither the parentheses nor the left-to-right grouping of the following expression guarantees what the value of `i` will be in either of the subexpressions:

```
( i++ +1 ) * ( 2 + i )
```

The compiler is free to evaluate the two sides of the multiplication in any order. If the initial value of `i` is zero, the whole expression could be evaluated as either of these two statements:

```
( 0 + 1 + 1 ) * ( 2 + 1 )
( 0 + 1 + 1 ) * ( 2 + 0 )
```

Exceptions resulting from side effects are discussed on page 109.

L-Value and R-Value Expressions

Expressions that refer to memory locations are called "l-value" expressions. An l-value represents a storage region's " locator" value, or a "left" value, implying that it can appear on the left of the equal sign (=). L-values are often identifiers.

Expressions referring to modifiable locations are called "modifiable l-values." A modifiable l-value cannot have an array type, an incomplete type, or a type with the **const** attribute. For structures and unions to be modifiable l-values, they must not have any members with the **const** attribute. The name of the identifier denotes a storage location, while the value of the variable is the value stored at that location.

An identifier is a modifiable l-value if it refers to a memory location and if its type is arithmetic, structure, union, or pointer. For example, if `ptr` is a pointer to a storage region, then `*ptr` is a modifiable l-value that designates the storage region to which `ptr` points.

Any of the following C expressions can be l-value expressions:

- An identifier of integral, floating, pointer, structure, or union type
- A subscript ([ ]) expression that does not evaluate to an array
- A member-selection expression (–> or **.**)
- A unary-indirection (*) expression that does not refer to an array
- An l-value expression in parentheses
- A **const** object (a nonmodifiable l-value)

The term "r-value" is sometimes used to describe the value of an expression and to distinguish it from an l-value. All l-values are r-values but not all r-values are l-values.

Microsoft Specific Microsoft C includes an extension to the ANSI C standard that allows casts of l-values to be used as l-values, as long as the size of the object does not change. (See "Type-Cast Conversions" on page 147 for more information.) The following example illustrates this feature:

```
char *p ;
short  i;
long l;

(long *) p = &l ;       /* Legal cast   */
(long) i = l ;          /* Illegal cast */
```

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za command-line option to disable these extensions.♦

Constant Expressions

A constant expression is evaluated at compile time, not run time, and can be used in any place that a constant may be used. The constant expression must evaluate to a constant that is in the range of representable values for that type. The operands of a constant expression can be integer constants, character constants, floating-point constants, enumeration constants, type casts, **sizeof** expressions, and other constant expressions.

Syntax

constant-expression :
  conditional-expression

conditional-expression :
  logical-OR-expression
  logical-OR-expression **?** *expression* **:** *conditional-expression*

expression :
  assignment-expression
  expression **,** *assignment-expression*

assignment-expression :
  conditional-expression
  unary-expression assignment-operator assignment-expression

assignment-operator : one of
  = *= /= %= += –= <<= >>= &= ^= |=

The nonterminals for struct declarator, enumerator, direct declarator, direct-abstract declarator, and labeled statement contain the *constant-expression* nonterminal.

An integral constant expression must be used to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a **case** constant.

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 "The Restricted Constant Expression" on page 204 for more information.)

Expression Evaluation

Expressions involving assignment, unary increment, unary decrement, or calling a function may have consequences incidental to their evaluation (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.

"Side effects" are changes caused by the evaluation of an expression. Side effects occur whenever the value of a variable is changed by an expression evaluation. All assignment operations have side effects. Function calls can 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.

Side Effects

The order of evaluation of expressions is defined by the specific implementation, except when the language guarantees a particular order of evaluation (as outlined in "Precedence and Order of Evaluation" on page 112). For example, side effects occur in the following function calls:

```
add( i + 1, i = j + 2 );
myproc( getc(), getc() );
```

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. Likewise, it is not possible to guarantee what characters are actually passed to the `myproc`. Since unary increment and decrement operations involve assignments, such operations can cause side effects, as shown in the following example:

```
x[i] = i++;
```

In this example, the value of `x` that is modified is unpredictable. The value of the subscript could be either the new or the old value of `i`. The result can vary under different compilers or different optimization levels.

Since C does not define the order of evaluation of side effects, both evaluation methods discussed above 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.

Sequence Points

Between consecutive "sequence points" an object's value can be modified only once by an expression. The C language defines the following sequence points:

- Left operand of the logical AND operator (**&&**). The left operand of the logical AND operator is completely evaluated and all side effects complete before continuing. If the left operand evaluates to false (0), the other operand is not evaluated.
- Left operand of the logical OR operator (||). The left operand of the logical OR operator is completely evaluated and all side effects complete before continuing. If the left operand evaluates to true (nonzero), the other operand is not evaluated.
- Left operand of the comma operator. The left operand of the comma operator is completely evaluated and all side effects complete before continuing. Both operands of the comma operator are always evaluated. Note that the comma separating arguments in a function call do not guarantee an order of evaluation.

- Function-call operator. All arguments to a function are evaluated and all side effects complete prior to entry to the function. No order of evaluation among the arguments is specified.
- First operand of the conditional operator. The first operand of the conditional operator is completely evaluated and all side effects complete before continuing.
- The end of a full initialization expression (that is, an expression that is not part of another expression).
- The expression in an expression statement. Expression statements consist of an optional expression followed by a semicolon (**;**). The expression is evaluated for its side effects and there is a sequence point following this evaluation.
- The controlling expression in a selection (**if** or **switch**) statement. The expression is completely evaluated and all side effects complete before the code dependent on the selection is executed.
- The controlling expression of a **while** or **do** statement. The expression is completely evaluated and all side effects complete before any statements in the next iteration of the **while** or **do** loop are executed.
- Each of the three expressions of a **for** statement. The expressions are completely evaluated and all side effects complete before any statements in the next iteration of the **for** loop are executed.
- The expression in a **return** statement. The expression is completely evaluated and all side effects complete before control returns to the calling function.

4.2 Operators

There are three types of operators. A unary expression consists of either a unary operator 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. A binary expression consists of two operands joined by a binary operator. A ternary expression consists of three operands joined by the ternary operator. C includes the following unary operators:

Symbol	Name
– ~ !	Negation and complement operators
* &	Indirection and address-of operators
sizeof	Size operator
+	Unary plus operator
++ ––	Unary increment and decrement operators

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
:>	Base operator

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. Each compound-assignment operator is a combination of another binary operator with the simple-assignment operator.

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. Precedence can also be described by the word "binding." Operators with a higher precedence are said to have tighter binding.

Table 4.1 summarizes the precedence and associativity (the order in which the operands are evaluated) of C operators, listing them in order of precedence from highest to lowest. Where several operators appear together, they have equal precedence and are evaluated according to their associativity. The operators in Table 4.1 are described in the sections beginning with "Postfix Operators" on page 116. The rest of this section gives general information about precedence and associativity.

Table 4.1 Precedence and Associativity of C Operators

Symbol[1]	Type of Operation	Associativity
[ ] ( ) . –> postfix ++ and postfix – – :>	Expression	Left to right

Table 4.1 Precedence and Associativity of C Operators (*continued*)

Symbol[1]	Type of Operation	Associativity
Prefix ++ and prefix – – **sizeof** & * + – ~ !	Unary	Right to left
typecasts	Unary	Right to left
* / %	Multiplicative	Left to right
+ –	Additive	Left to right
<< >>	Bitwise shift	Left to right
< > <= >=	Relational	Left to right
== !=	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[2]	Right to left
,	Sequential evaluation	Left to right

[1] Operators are listed in descending order of precedence. If several operators appear on the same line or in a group, they have equal precedence.

[2] All simple and compound-assignment operators have equal precedence.

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 multiplication (*), addition (+), or binary-bitwise (**&** | ^) operator at the same level. Order of operations is not defined by the language. The compiler is free to evalaute such expressions in any order, if the compiler can guarantee a consistent result.

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 set of 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.) Sequence points are discussed on page 110.

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. This is called "short-circuit" evaluation. 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).

The following list shows how the compiler automatically binds several sample expressions:

Expression	Automatic Binding
`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 (`||`) has higher 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. If not decrementing `s` would cause a problem in your program, `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 diagnostic message at compile time:

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 )
```

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." Conversion of an operand value to a compatible type causes no change to its value.

The arithmetic conversions summarized below are called "usual arithmetic conversions." These steps are applied only for binary operators that expect arithmetic type and only if the two operands do not have the same type. The purpose is to yield a common type which is also the type of the result. To determine which conversions actually take place, the compiler applies the following algorithm to binary operations in the expression. The steps below are not a precedence order.

1. If either operand is of type **long double**, the other operand is converted to type **long double**.
2. If the above condition is not met and either operand is of type **double**, the other operand is converted to type **double**.
3. If the above two conditions are not met and either operand is of type **float**, the other operand is converted to type **float**.
4. If the above three conditions are not met (none of the operands are of floating types), then integral conversions are performed on the operands as follows:
   a. If either operand is of type **unsigned long**, the other operand is converted to type **unsigned long**.
   b. If the above condition is not met and either operand is of type **long** and the other of type **unsigned int**, the operand of type **unsigned int** is converted to type **long** (in 16-bit compilations) or both operands are converted to type **unsigned long** (in 32-bit compilations).
   c. If the above two conditions are not met, and either operand is of type **long**, the other operand is converted to type **long**.
   d. If the above three conditions are not met, and either operand is of type **unsigned int**, the other operand is converted to type **unsigned int**.
   e. If none of the above conditions are met, both operands are converted to type **int**.

The following code illustrates these conversion rules:

```
float   fVal;
double  dVal;
int   iVal;
unsigned long ulVal;

dVal = iVal * ulVal; /* iVal converted to unsigned long
                      * Uses step 4.a.
                      * Result of multiplication converted to double
                      */
dVal = ulVal + fVal; /* ulVal converted to float
                      * Uses step 3.
                      * Result of addition converted to double
                      */
```

Postfix Operators

The postfix operators have the highest precedence (the tightest binding) in expression evaluation.

Syntax

postfix-expression :
  primary-expression
  postfix-expression [*expression*]
  postfix-expression (*argument-expression-list* $_{\text{opt}}$)
  postfix-expression **.** *identifier*
  postfix-expression **–>** *identifier*
  postfix-expression **++**
  postfix-expression **––**
  postfix-expression **:>** *expression* /* Microsoft-specific */

Operators in this precedence level are the array subscripts, function calls, structure and union members, and postfix increment and decrement operators.

One-Dimensional Arrays

A postfix expression followed by an expression in square brackets ([ ]) is a subscripted representation of an element of an array object. A subscript expression represents the value at the address that is *expression* positions beyond *postfix-expression* when expressed as

postfix-expression [*expression*]

Usually, the value represented by *postfix-expression* is a pointer value, such as an array identifier, and *expression* 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 *postfix-expression* position and the pointer value could be in the brackets in the *expression*, or "subscript," position. For example, this code is legal:

```
int sum, *ptr, a[10];

int main()
{
    ptr = a;
    sum = 4[ptr];
}
```

Subscript expressions are generally used to refer to array elements, but you can apply a subscript to any pointer. Whatever the order of values, *expression* must be enclosed in brackets ([ ]).

The subscript expression is evaluated by adding the integral value to the pointer value, then applying the indirection operator (*) to the result. (See "Indirection and Address-of Operators" on page 122 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 "Additive Operators" on page 128), 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 number of bytes defined as the length of an **int** item. 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 ]`).

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 Arrays

A subscript expression can also have multiple subscripts, as follows:

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

Subscript expressions associate from left to right. The leftmost 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 examples 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.

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];
```

A reference to the `prop` array looks like this:

```
i = prop[0][0][1];
```

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

```
i = prop[2][1][3];
```

This statement is 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.

These next two examples show cases where the indirection operator is not applied.

```
ip = prop[2][1];

ipp = prop[2];
```

In the first of these statements, the expression `prop[2][1]` is a valid reference to the three-dimensional array `prop`; it refers to a 6-element array (declared above). Since the pointer value addresses an array, the indirection operator is not applied.

Similarly, the result of the expression `prop[2]` in the statement `ipp = prop[2];` is a pointer value addressing a two-dimensional array.

Function Calls

A "function call" is an expression that includes the name of the function being called or the value of a function pointer and, optionally, the arguments being passed to the function.

Syntax

postfix-expression :
  postfix-expression (*argument-expression-list* $_{\text{opt}}$)

argument-expression-list :
  assignment-expression
  argument-expression-list , *assignment-expression*

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

A function-call expression has the value and type of the function's return value. A function cannot return an object of array type. 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. (See "Function Calls" on page 183 for more information.)

Structure and Union Members

A "member-selection expression" refers to members of structures and unions. Such an expression has the value and type of the selected member.

Syntax

postfix-expression **.** *identifier*
postfix-expression **–>** *identifier*

This list describes the two forms of the member-selection expressions:

1. In the first form, *postfix-expression* represents a value of **struct** or **union** type, and *identifier* names a member of the specified structure or union. The value of

the operation is that of *identifier* and is an l-value if *postfix-expression* is an l-value. See "L-Value and R-Value Expressions" on page 107 for more information.

2. In the second form, *postfix-expression* represents a pointer to a structure or union, and *identifier* names a member of the specified structure or union. The value is that of *identifier* and is an l-value.

The two forms of member-selection expressions have similar effects.

In fact, an expression involving the member-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. Therefore,

expression –> *identifier*

is equivalent to

(**expression*) **.** *identifier*

when *expression* is a pointer value.

The following examples refer to this structure declaration. See page 122 for information about the indirection operator (*) used in these examples.

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

A member-selection expression for the `item` structure looks like this:

```
item.sp = &item;
```

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

```
(item.sp)->a = 24;
```

In this example, the pointer expression `item.sp` is used with the member-selection operator (–>) to assign a value to the member `a`.

```
list[8].b = 12;
```

This statement shows how to select an individual structure member from an array of structures.

Postfix Increment and Decrement Operators

Operands of the postfix increment and decrement operators are scalar types that are modifiable l-values.

Syntax

postfix-expression :
  postfix-expression **++**
  postfix-expression **––**

The result of the postfix increment or decrement operation is the value of the operand. After the result is obtained, the value of the operand is incremented (or decremented). The following code illustrates the postfix increment operator.

```
if( var++ > 0 )
    *p++ = *q++;
```

In this example, the variable `var` is compared to 0, then incremented. If `var` was positive before being incremented, the next statement is executed. First, the value of the object pointed to by `q` is assigned to the object pointed to by `p`. Then, `q` and `p` are incremented.

Base Operator (Microsoft Specific)

The base operator (**:>**) is a Microsoft extension added to support based addressing. The base operator combines a memory segment address with an address that can be dereferenced with the indirection (*****) operator. It has the form

segvar **:>** *offset*

The *segvar* can be any expression that evaluates to a valid memory segment address. You can use a variable or expression of type **__segment**, or create your own segments using the **__segname** operator. The *offset* can be a pointer with the form

type **__based(void)** *

or any 16-bit expression.

You can use the base operator to create a pointer that can address any memory location. See "Based Pointers" on page 79 for more information.

32-Bit Specific

The base operator is not supported for 32-bit targets.♦

Unary Operators

Unary operators appear before their operand and associate from right to left.

Syntax

unary-expression :
  postfix-expression
  ++ *unary-expression*
  –– *unary-expression*
  unary-operator cast-expression
  sizeof *unary-expression*
  sizeof (*type-name*)

unary-operator : one of
  & * + – ~ !

Prefix Increment and Decrement Operators

The unary operators (++ and ––) are called "prefix" increment or decrement operators when the increment or decrement operators appear before the operand. Postfix increment and decrement has higher precedence than prefix increment and decrement operators. The operand must have integral, floating, or pointer type and must be a modifiable l-value expression (an expression without the **const** attribute). The result is not an l-value.

When the operator appears before its operand, the operand is incremented or decremented and its new value is the result of the expression.

An operand of integral or floating type is incremented or decremented by the integer value 1. The type of the result 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.

This example illustrates the unary decrement operator:

```
if( line[--i] != '\n' )
    return;
```

In this example, the variable `i` is decremented before it is used as a subscript to `line`.

Indirection and Address-of Operators

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 addressed by the operand; that is, the value at the address to which its operand points. The type of the result is the type that the operand addresses.

If the operand points to a function, the result is a function designator. If it points to a storage location, the result is an l-value designating the storage location.

If the pointer value is invalid, the result is undefined. The following list includes some of the most common conditions that invalidate a pointer value.

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

The address-of operator (**&**) gives the address of its operand. The operand of the address-of operator can be either a function designator or an l-value that designates an object that is not a bit field and is not declared with the **register** storage-class specifier.

The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

The address-of operator can only be applied to variables with fundamental, structure, or union types that are declared at the file-scope 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.

The following examples use these declarations:

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

This statement uses the address-of operator.

```
pa = &a[5];
```

The address-of operator (**&**) takes the address of the sixth element of the array `a`. The result is stored in the pointer variable `pa`.

```
x = *pa;
```

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

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

This example prints the word `True`, demonstrating that the result of applying the indirection operator to the address of `x` is the same as `x`.

```
int roundup( void );     /* Function declaration */

int  *proundup  = roundup;
int  *pround    = &roundup;
```

Once the function `roundup` is declared, two pointers to `roundup` are declared and initialized. The first pointer, `proundup`, is initialized using only the name of the function, while the second, `pround`, uses the address-of operator in the initialization. The initializations are equivalent.

Unary Arithmetic Operators

The C unary plus, arithmetic-negation, complement, and logical-negation operators are discussed in the following list:

Operator	Description
+	The unary plus operator preceding an expression in parentheses forces the grouping of the enclosed operations. It is used with expressions involving more than one associative or commutative binary operator. The operand must have arithmetic type. The result is the value of the operand. An integral operand undergoes integral promotion. The type of the result is the type of the promoted operand.
–	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.
~	The bitwise-complement (or bitwise-NOT) 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.
!	The logical-negation (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.

The following examples illustrate these operators:

```
short x = 987;
    x = -x;
```

In the example above, the new value of `x` is the negative of 987, or –987.

```
unsigned short y = 0xAAAA;
    y = ~y;
```

In this example, the new value assigned to `y` is the one's complement of the unsigned value 0xAAAA, or 0x5555.

```
if( !(x < y) )
```

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).

The sizeof Operator

The **sizeof** operator gives the amount of storage, in bytes, required to store an object of the type of the operand. This operator allows you to avoid specifying machine-dependent data sizes in your programs.

Syntax

sizeof *unary-expression*
sizeof (*type-name*)

The operand is either an identifier that is a *unary-expression*, or a type-cast expression (that is, a type specifier enclosed in parentheses). The *unary-expression* cannot represent a bit-field object, an incomplete type, or a function designator. The result is an unsigned integral constant. The standard header STDDEF.H defines this type as **size_t**.

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 number of bytes in the structure or union, including internal and trailing padding. 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.

If an unsized array is the last element of a structure, the **sizeof** operator returns the size of the structure without the array.

```
buffer = calloc(100, sizeof (int) );
```

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

```
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 this example, `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** value, `string_no` cannot be modified.

Cast Operators

A type cast provides a method for explicit conversion of the type of an object in a specific situation.

Syntax

cast-expression :
  unary-expression
  (*type-name*) *cast-expression*

The compiler treats *cast-expression* as type *type-name* after a type cast has been made. 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 "Assignment Conversions" on page 141. Additional restraints on casts may result from the actual sizes or representation of specific types. See "Storage of Basic Types" on page 98 for information on actual sizes of integral types. For more information on type casts, see "Type-Cast Conversions" on page 147.

Multiplicative Operators

The multiplicative operators perform multiplication (*), division (/), and remainder (%) operations.

Syntax

multiplicative-expression :
  cast-expression
  multiplicative-expression * *cast-expression*
  multiplicative-expression / *cast-expression*
  multiplicative-expression % *cast-expression*

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.

The C multiplicative operators are described below:

Operator	Description
*	The multiplication operator causes its two operands to be multiplied.
/	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: ▪ The result of division by 0 is undefined according to the ANSI standard. The Microsoft C compiler generates an error at compile- or run-time. ▪ If both operands are positive or unsigned, the result is truncated toward 0. ▪ If either operand is negative, whether the result of the operation is the largest integer less than or equal to the algebraic quotient or is the smallest integer greater than or equal to the algebraic quotient is implementation defined. (See the Microsoft Specific section below.)
%	The result of the remainder operator is the remainder when the first operand is divided by the second. When the division is inexact, the result is determined by the following rules: ▪ If the right operand is zero, the result is undefined. ▪ If either or both operands are positive, the result is positive. ▪ If either operand is negative and the result is inexact, the result is implementation defined. (See the Microsoft Specific section below.)

Microsoft Specific

In division where either operand is negative, the direction of truncation is toward 0.

If either operation is negative in division with the remainder operator, the result has the same sign as the dividend (the first operand in the expression).♦

The declarations shown below are used for the following examples:

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

This statement uses the multiplication operator:

```
y = x * i;
```

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

```
n = i / j;
```

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

```
n = i % j;
```

This statement assigns `n` the integer remainder, 1, when 10 is divided by 3.

Microsoft Specific The sign of the remainder is the same as the sign of the dividend. For example,

```
50 % -6 = 2
-50 % 6 = -2
```

In each case, `50` and `2` have the same sign.♦

Additive Operators

The additive operators perform addition (+) and subtraction (–).

Syntax

additive-expression :
  multiplicative-expression
  additive-expression + *multiplicative-expression*
  additive-expression – *multiplicative-expression*

Note Although the syntax for *additive-expression* includes *multiplicative-expression*, this does not imply that expressions using multiplication are required. See the syntax in Appendix A for *multiplicative-expression*, *cast-expression*, and *unary-expression*.

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 be either 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 and therefore is the same as array indexing (see "One-Dimensional Arrays" on page 116 and "Multidimensional Arrays" on page 117). If the sum pointer points outside the array, except at the first location beyond the high end, the result is undefined. See "Pointer Arithmetic" later in this section for more information.

Subtraction (–)

The subtraction operator (–) subtracts the second operand from the first. Both operands can be either integral or floating types, or one operand can be a pointer and the other 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 type **ptrdiff_t** in the standard include file STDDEF.H. 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 in "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.

Using the Additive Operators

The following examples, which illustrate the addition and subtraction operators, use these declarations:

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

These statements are equivalent:

```
px = &x[4 + i];
px = &x[4] + i;
```

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]`.

```
j = &x[i] - &x[i-2];
```

In this example, 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.

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 filled by the same type of identifiers. 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 the reference information on these processors or reference books on segmented architecture for more information.

Note The results of pointer subtraction when both operands are pointers to **short** may overflow the intermediate result. To avoid this, cast the pointers to pointers to **long**. For subtraction of **__huge** operands, cast the difference to **unsigned long** to get the correct result.

Bitwise Shift Operators

The shift operators shift their first operand left (<<) or right (>>) by the number of positions the second operand specifies.

Syntax

shift-expression :
  additive-expression
  shift-expression << *additive-expression*
  shift-expression >> *additive-expression*

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. For left-shift operators without overflow, the statement

```
expr1 << expr2
```

is equivalent to multiplication by 2^{expr2}. For right-shift operators,

```
expr1 >> expr2
```

is equivalent to division by 2^{expr2} if `expr1` is unsigned or has a nonnegative value.

The result of a shift operation is undefined if the second operand is negative, or if the right operand is greater than or equal to the width in bits of the promoted left operand.

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.

```
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`.

Shifting a negative value to the right yields the negative of half the absolute value, rounded down. For example, –253 (binary 11111111 00000011) on a 16-bit computer shifted right one bit produces –127 (binary 11111111 10000001). A positive 253 shifts right to produce +126.

Right shifts preserve the sign of the number. When a signed integer shifts right, the most-significant bit remains set as it had been. When an unsigned integer shifts right, the most-significant bit is cleared. Thus if 0xF000 is signed, a right shift produces 0xF800. If 0xF000 is unsigned, the result is 0x7800.

Shifting a positive number right 16 times using the 16-bit computer produces 0x0000. Shifting a negative number right 16 times produces 0xFFFF.

Relational and Equality Operators

The binary relational and equality 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**.

Syntax

relational-expression :
  shift-expression
  relational-expression < *shift-expression*
  relational-expression > *shift-expression*
  relational-expression <= *shift-expression*
  relational-expression >= *shift-expression*

equality-expression :
  relational-expression
  equality-expression == *relational-expression*
  equality-expression != *relational-expression*

The relational and equality 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 first four operators in the list above have a higher precedence than the equality operators (== and !=). See the precedence information in Table 4.1.

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 the relational and equality operators:

- Both operands of any relational or equality operator can be pointers to the same type. For the equality (==) and inequality (!=) operators, the result of the comparison indicates whether the two pointers address the same memory location. For the other relational operators (<, >, <=, and >=), the result of the comparison indicates the relative position of the two memory addresses of the objects pointed to. Relational operators compare only offsets.

Pointer comparison is defined only for parts of the same object. If the pointers refer to members of an array, the comparison is equivalent to comparison of the corresponding subscripts. The address of the first array element is "less than" the address of the last element. In the case of structures, pointers to structure members declared later are "greater than" pointers to members declared earlier in the structure. Pointers to the members of the same union are equal.

- A pointer value can be compared to the constant value 0 for equality (= =) or inequality (!=) pointer. A pointer with a value of 0 is called a "null" pointer; that is, it does not point to a valid memory location.
- The equality operators follow the same rules as the relational operators, but permit additional possibilities: a pointer may be compared to a constant integral expression with value 0, or to a pointer to **void**. If two pointers are both null pointers, they compare as equal. Equality operators compare both segment and offset.

The examples below illustrate relational and equality operators.

```
int x = 0, y = 0;
if( x < y )
```

Because `x` and `y` are equal, the expression in this example yields the value 0.

```
char array[10];
char *p;

for ( p = array; p < &array[10]; p++ )
    *p = '\0';
```

The fragment in this example sets each element of `array` to a null character constant.

```
enum color { red, white, green } col;
    .
    .
    .
    if ( col == red )
    .
    .
    .
```

These statements declare an enumeration variable named `col` with the tag `color`. At any time, the variable may contain an integer value of 0, 1, or 2, which represents one of the elements of the enumeration set `color`: the color red, white, or green, respectively. If `col` contains 0 when the **if** statement is executed, any statements depending on the **if** will be executed.

Bitwise Operators

The bitwise operators perform bitwise-AND (**&**), bitwise-exclusive-OR (**^**), and bitwise-inclusive-OR (**|**) operations.

Syntax

AND-expression :
  equality-expression
  AND-expression **&** *equality-expression*

exclusive-OR-expression :
  AND-expression
  exclusive-OR-expression **^** *AND-expression*

inclusive-OR-expression :
  exclusive-OR-expression
  inclusive-OR-expression **|** *exclusive-OR-expression*

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.

The C bitwise operators are described below:

Operator	Description
&	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.
^	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.
|	The bitwise-inclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

These declarations are used for the following three examples:

```
short i = 0xAB00;
short j = 0xABCD;
short n;

n = i & j;
```

The result assigned to `n` in this first example is the same as `i` (0xAB00 hexadecimal).

```
n = i | j;

n = i ^ j;
```

The bitwise-inclusive OR in the second example results in the value 0xABCD (hexadecimal), while the bitwise-exclusive OR in the third example produces 0xCD (hexadecimal).

Microsoft Specific

The results of bitwise operation on signed integers is implementation-defined according to the ANSI standard. For the Microsoft C compiler, bitwise operations on signed integers work the same as bitwise operations on unsigned integers. For example `-16 & 99` can be expressed in binary as

```
  11111111 11110000
& 00000000 01100011
  -----------------
  00000000 01100000
```

The result of the bitwise AND is 96 decimal.♦

Logical Operators

The logical operators perform logical-AND (**&&**) and logical-OR (||) operations.

Syntax

logical-AND-expression :
  inclusive-OR-expression
  logical-AND-expression **&&** *inclusive-OR-expression*

logical-OR-expression :
  logical-AND-expression
  logical-OR-expression || *logical-AND-expression*

Logical operators do not perform the usual 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 result's type is **int**.

The C logical operators are described below:

Operator	Description
&&	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.

Operator	Description
||	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.

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. This is called "short-circuit evaluation." There is a sequence point after the first operand. See "Sequence Points" on page 110 for more information.

The following examples illustrate the logical operators:

```
int w, x, y, z;

if( x < y && y < z )
    printf( "x is less than z\n" );
```

In this example, 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 has side effects that are being relied on for some other reason.

```
printf( "%d" , (x == w || x == y || x == z) );
```

In this example, 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.

Conditional Operator

C has one ternary operator: the conditional operator (**? :**).

Syntax

conditional-expression :
  logical-OR-expression
  logical-OR-expression **?** *expression* **:** *conditional-expression*

The *logical-OR-expression* must have integral, floating, or pointer type. It is evaluated in terms of its equivalence to 0. A sequence point follows *logical-OR-expression.* Evaluation of the operands proceeds as follows:

- If *logical-OR-expression* is not equal to 0, *expression* is evaluated. The result of evaluating the expression is given by the nonterminal *expression.* (This means *expression* is evaluated only if *logical-OR-expression* is true.)
- If *logical-OR-expression* equals 0, *conditional-expression* is evaluated. The result of the expression is the value of *conditional-expression.* (This means *conditional-expression* is evaluated only if *logical-OR-expression* is false.)

Note that either *expression* or *conditional-expression* is evaluated, but not both.

The type of the result of a conditional operation depends on the type of the *expression* or *conditional-expression* operand, as follows:

- If *expression* or *conditional-expression* 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 *expression* and *conditional-expression* 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 *expression* or *conditional-expression* is a pointer and the other operand is a constant expression with the value 0, the type of the result is the pointer type.

In the type comparison for pointers, any type qualifiers (**const** or **volatile**) in the type to which the pointer points are insignificant, but the result type inherits the qualifiers from both components of the conditional.

The following examples show uses of the conditional operator:

```
j = ( i < 0 ) ? ( -i ) : ( i );
```

This example 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`.

```
void f1( void );
void f2( void );
int x;
int y;
    .
    .
    .
( x == y ) ? ( f1() ) : ( f2() );
```

In this example, 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.

Assignment Operators

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 must be a modifiable l-value. After the assignment, an assignment expression has the value of the left operand but is not an l-value.

Syntax

assignment-expression :
    conditional-expression
    unary-expression assignment-operator assignment-expression

assignment-operator : one of
    = *= /= %= += –= <<= >>= &= ^= |=

The assignment operators in C can both transform and assign values in a single operation. C provides the following assignment operators:

Operator	Operation Performed
=	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, and the value is stored in the left operand after the assignment has taken place. The left operand must not be an array, a function, or a constant. The specific conversion path, which depends on the two types, is outlined in detail in "Type Conversions" on page 141.

Simple Assignment

The simple-assignment operator assigns its right operand to its left operand. The value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand. The conversion rules for assignment apply (see "Assignment Conversions" on page 141).

```
double x;
int y;

x = y;
```

In this example, the value of `y` is converted to type **double** and assigned to `x`.

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.

```
#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. (For more information, see "The #define Directive" on page 193.)

Sequential-Evaluation Operator

The sequential-evaluation operator, also called the "comma operator," evaluates its two operands sequentially from left to right.

Syntax

expression :
  assignment-expression
  expression **,** *assignment-expression*

The left operand of the sequential-evaluation operator is evaluated as a **void** expression. The result of the operation has the same value and type as the right operand. Each operand can be of any type. The sequential-evaluation operator does not perform type conversions between its operands, and it does not yield an l-value. There is a sequence point after the first operand, which means all side effects from the evaluation of the left operand are completed before beginning evaluation of the right operand. See page 110 for information on sequence points.

The sequential-evaluation operator is typically used to evaluate two or more expressions in contexts where only one expression is allowed.

Commas may be used as separators in some contexts. However, 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.

This example illustrates the sequential-evaluation operator:

```
for ( i = j = 1; i + j < 20; i += i, j-- );
```

In this example, 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.

```
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`.

4.3 Type Conversions

Type conversions depend on the specified operator and the type of the operand or operators. Type conversions are performed in the following cases:

- When a value of one type is assigned to a variable of a different type or an operator converts the type of its operand or operands before performing an operation
- When a value of one type is explicitly cast to a different type
- When a value is passed as an argument to a function or when a type is returned from a function

This section outlines the rules for each kind of conversion.

A character, a short integer, or an integer bit field, all either signed or not, or an object of enumeration type, can be used in an expression wherever an integer can be used. If an **int** can represent all the values of the original type, then the value is converted to **int**; otherwise, it is converted to **unsigned int**. This process is called "integral promotion." Integral promotions preserve value. That is, the value after promotion is guaranteed to be the same as prior to the promotion. See "Usual Arithmetic Conversions" on page 115 for more information.

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 method used depends on the types involved in the assignment, as described in "Usual Arithmetic Conversion" on page 115 and in the following sections.

Type qualifiers do not affect the allowability of the conversion although a **const** l-value cannot be used on the left side of the assignment.

Conversions from Signed Integral Types

When a signed integer is converted to an unsigned integer with equal or greater size and the value of the signed integer is not negative, the value is unchanged. The conversion is made by sign-extending the signed integer. A signed integer is converted to a shorter signed integer by truncating the high-order bits. The result is interpreted as an unsigned value, as shown in this example.

```
int i = -3;
unsigned u;

u = i;
printf( "%u\n", u );  /* Prints 65533 on 16-bit computers */
```

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 4.2 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 4.3 for the **unsigned char** type apply instead of the conversions in Table 4.2.

Table 4.2 Conversions from Signed Integral Types

From	To	Method
char[1]	**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**
char	**long 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**
short	**long 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.

Table 4.2 Conversions from Signed Integral Types (*continued*)

From	To	Method
long	**double**	Represent as **double**. If **long** cannot be representèd exactly as a **double**, some precision is lost.
long	**long double**	Represent as **double**. If **long** cannot be represented exactly as a **double**, some precision is lost.

[1] All **char** entries assume that the **char** type is signed by default.

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. For the Microsoft C compiler, an integer is the same as a **short** for 16-bit targets, and is equivalent to a **long** for 32-bit targets.

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 the value with integral type is demoted to a signed integer with smaller size, or an unsigned integer is converted to its corresponding signed integer, the value is unchanged if it can be represented in the new type. However, the value it represents changes if the sign bit is set. The results in the following example are true for 16-bit computers.

```
int j;
unsigned k = 65533;

j = k;
printf( "%d\n", j );   /* Prints -3 */
```

If it cannot be represented, the result is implementation-defined. See "Type-Cast Conversions" on page 147 for information on the Microsoft C compiler's handling of demotion of integers. The same behavior results from integer conversion or from type casting the integer.

Unsigned values are converted in a way that preserves their value and is not representable directly in C. The only exception is a conversion from **unsigned long** to **float** which loses at most the low-order bits. Otherwise value is preserved, signed or unsigned. When a value of integral type is converted to floating, and the value is outside the range representable, the result is undefined. (See "Storage of Basic Types" on page 98 for information about the range for integral and floating-point types.)

Table 4.3 summarizes conversions from unsigned integral types.

Table 4.3 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**
unsigned char	**double**	Convert to **long**; convert **long** to **double**
unsigned char	**long 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 short	**long 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 directly to **double**
unsigned long	**long double**	Convert to **long**; convert **long** to **double**

The **unsigned int** type is equivalent either to the **unsigned short** type or to the **unsigned long** type, depending on the target environment. Conversion of an **unsigned int** value proceeds in the same way as conversion of an **unsigned short** or an **unsigned long**, whichever is appropriate. For the Microsoft C compiler, an integer is the same as a **short** for 16-bit targets and is equivalent to a **long** for 32-bit targets. Conversions from **unsigned long** values to **float** are not accurate if the value being converted is larger than the maximum positive signed **long** value.

Conversions from Floating-Point Types

A **float** value converted to a **double** or **long double**, or a **double** converted to a **long double**, undergoes no change in value. A **double** value converted to a **float** value is represented exactly, if possible. Precision may be lost if the value cannot be represented exactly. If the result is out of range, the behavior is undefined. See "Floating-Point Constants" on page 10 for the range of floating-point types.

A floating value is converted to an integral value by first converting to a **long**, then from the **long** value to the specific integral value, as described below in Table 4.4. 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.

Microsoft Specific

When converting a **double** or **long double** floating-point number to a smaller floating-point number, the value of the floating-point variable is truncated toward zero when an underflow occurs. An overflow causes a run-time error.♦

Table 4.4 summarizes conversions from floating types.

Table 4.4 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	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
float	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
float	**double**	Change internal representation
float	**long 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	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
double	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
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.
long double	**char**	Convert to **float**; convert **float** to **char**

Table 4.4 Conversions from Floating-Point Types (*continued*)

From	To	Method
long double	**short**	Convert to **float**; convert **float** to **short**
long double	**long**	Truncate at decimal point. If result is too large to be represented as **long**, result is undefined.
long double	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
long double	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
long 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.
long double	**double**	The **long double** value is treated as **double**.

Note Conversions from **float**, **double**, or **long double** values to **unsigned long** are not accurate if the value being converted is larger than the maximum positive **long** value.

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 an object may be converted to a pointer to an object whose type requires less or equally strict storage alignment, and back again without change.

A pointer to **void** may be converted to or from a pointer to any type, without restriction or loss of information. If the result is converted back to the original type, the original pointer is recovered.

If a pointer is converted to another pointer with the same type but having different or additional qualifiers, the new pointer is the same as the old except for restrictions imposed by the new qualifier.

A pointer value can also be converted to an integral value. The conversion path depends on the size of the pointer and the size of the integral type, according to the following rules:

- If the size of the pointer is 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.

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 4.2 and 4.3. It is then treated as a pointer value.

An integral constant expression with value 0 or such an expression cast to type **void** * may be converted by a type cast, by assignment, or by comparison to a pointer of any type. This produces a null pointer that is equal to another null pointer of the same type, but this null pointer is not equal to any pointer to a function or to an object. Integers other than the constant 0 may be converted to pointer type, but the result is not portable.

See "Special Keywords in Declarators" on page 55 for information about conversions on pointers made with the **__near**, **__far**, and **__huge** keywords.

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 target environment.

Microsoft Specific

For the Microsoft C compiler, an integer is the same as a **short** for 16-bit targets and is equivalent to a **long** for 32-bit targets.

No conversions between structure or union types are allowed.

Any value may be converted to type **void**, but the result of such a conversion can be used only in a context where an expression value is discarded, such as in an expression statement.

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 the next section.♦

Type-Cast Conversions

You can use type casts to explicitly convert types.

Syntax

cast-expression :
  unary expression
  (*type-name*) *cast-expression*

type-name :
  specifier-qualifier-list abstract-declarator $_{\text{opt}}$

The *type-name* is a type and *operand* is a value to be converted to that type. An expression with a type cast is not an l-value. The operand is converted as though it had been assigned to a variable of type *type-name.* The conversion rules for assignments (outlined in "Assignment Conversions" on page 141) apply to type casts as well.

Any identifier may be cast to **void** type. However, if the type specified in a type-cast expression is not **void**, then the identifier being cast to that type cannot be a **void** expression. Any expression can be cast to **void**, but an expression of type **void** cannot be cast to any other type. For example, a function with **void** return type cannot have its return cast to another type.

Note that a **void** * expression has a type pointer to **void**, not type **void**. 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 l-value, so no assignment can be made to a type-cast object.

Microsoft Specific

A type cast can be an l-value expression as long as the size of the identifier does not change. See page 107 for information on l-value expressions.♦

You can convert an expression to type **void** with a cast, but the resulting expression can be used only where a value is not required. An object pointer converted to **void** * and back to the original type will return to its original value.

Table 4.5 shows the types that can be cast to any given type.

Table 4.5 Legal Type Casts

Destination Types	**Potential Sources**
Integral types	Any integer type or floating-point type, or pointer to an object
Floating-point	Any arithmetic type
A pointer to an object, or (**void** *)	Any integer type, (**void** *), a pointer to an object, or a function pointer
Function pointer	Any integral type, a pointer to an object, or a function pointer
A structure, union, or array	None
Void type	Any type

Microsoft Specific When a **long** integer is cast to a **short**, or a **short** is cast to a **char** (demotion), the least-significant bytes are retained.

For example, this statement

```
short x = (short)x12345678L;
```

assigns the value 0x5678 to `x`, and this statement

```
char y = (char)0x1234;
```

assigns the value 0x34 to `y`.

On a 16-bit computer, near pointers are the same size as **short** integers; casting near pointers to **short** (or **short** to near pointers) has no effect on the value. Far pointers and huge pointers are the same size as **long** integers. Casting far or huge pointers to **long** (or **long** to far or huge pointers) has no effect on the value.

When a near pointer on a 16-bit computer is cast to long, the 16-bit value is "normalized," which means the segment (usually DS) and offset are combined to produce a 32-bit memory location. When a far or huge pointer is cast to **short**, the value is truncated to a **short**.

The compiler normalizes based pointers when necessary, unless the based pointer is a constant zero, in which case it is assumed to be a null pointer. See page 79 for more information on based pointers.

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

For example, casting an **unsigned long** (with 32 bits of precision) to a **float** (whose mantissa has 23 bits of precision) rounds the number to the nearest multiple of 256. The **long** values in the range of 4,294,966,913 to 4,294,967,167 are all rounded to the **float** value 4,294,967,040. ♦

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 includes declared argument types, the compiler performs type checking (see Chapter 6, "Functions").

If no function prototype is present, 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**.

Microsoft Specific

If the special keywords **__near**, **__far**, and **__huge** are used, implicit conversions may also be made on pointer values passed to functions. You can override these implicit conversions by providing function prototypes, which also allow the compiler to perform type checking.

Statements

Chapter 5

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 the following C statements in alphabetical order:

break statement
compound statement
continue statement
do statement
expression statement
for statement
goto and labeled statements
if statement
null statement
return statement
switch statement
while statement

5.1 Overview

C statements consist of tokens, expressions, and other statements. A statement that forms a component of another statement is called the "body" of the enclosing statement. Each of the statement types given by the following syntax is discussed later in this chapter.

Syntax

statement :
  labeled-statement
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement

Frequently the statement body is a "compound statement." A compound statement is made up of other statements that can include keywords. The compound statement is delimited by braces ({ }). All other C statements end with a semicolon (;). The semicolon is a statement terminator.

The expression statement contains a C expression that can contain the arithmetic or logical operators introduced in Chapter 4. The null statement is an empty statement.

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 discussed with **goto**. For more information, see "The goto and Labeled Statements" on page 157.

5.2 The break Statement

The **break** statement terminates the execution of the nearest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. Control passes to the statement that follows the terminated statement.

Syntax

jump-statement :
  break;

The **break** statement is frequently used to terminate the processing of a particular case within a **switch** statement. An error is generated if there is no enclosing iterative or switch 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 elsewhere out of the nested structure.

This example illustrates the **break** statement:

```
for ( i = 0; i < LENGTH; i++ )  /* Execution returns here when */
{                                /* break statement is executed */
    for( j = 0; j < WIDTH; j++)
    {
        if( lines[i][j] == '\0' )
        {
            lengths[i] = j;
            break;
        }
    }
}
```

The 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]`. The variable `j` is not incremented when **break** causes the exit from the interior loop. 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`.

5.3 The Compound Statement

A compound statement (also called a “block”) typically appears as the body of another statement, such as the **if** statement. Chapter 3, “Declarations and Types,” describes the form and meaning of the declarations that can appear at the head of a compound statement.

Syntax

compound-statement :
 { *declaration-list* $_{\text{opt}}$ *statement-list* $_{\text{opt}}$ }

declaration-list :
 declaration
 declaration-list declaration

statement-list :
 statement
 statement-list statement

If there are declarations, they must come before any statements. The scope of each identifier declared at the beginning of a compound statement extends from its declaration point to the end of the block. It is visible throughout the block unless a declaration of the same identifier exists in an inner block.

Identifiers in a compound statement are presumed **auto** unless explicitly declared otherwise with **register, static**, or **extern**, except functions which can only be **extern**. You can leave off the **extern** specifier in function declarations and the function will still be **extern**.

Storage is not allocated and initialization is not permitted if a variable or function is declared in a compound statement with storage class **extern**. The declaration refers to an external variable or function defined elsewhere.

Variables declared in a block with the **auto** or **register** keyword are reallocated and, if necessary, initialized each time the compound statement is entered. These variables are not defined after the compound statement is exited. If a variable declared inside of a block has the **static** attribute, the variable is initialized when program execution begins and keeps its value throughout the program. See “Storage Classes” on page 43 for information about **static**.

This example illustrates a compound statement:

```
if( i > 0 )
{
    line[i] = x;
    x++;
    i--;
}
```

In this example, if `i` is greater than 0, all of the statements inside the compound statement are executed in order.

5.4 The continue Statement

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. A typical use of the **continue** statement is to return to the start of a loop from within a deeply nested loop.

Syntax

jump-statement :
  continue;

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.
- A **continue** statement in a **for** statement causes the first expression of the **for** statement to be evaluated. Then the compiler reevaluates the conditional expression and, depending on the result, either terminates or iterates the statement body. For more information on the **for** statement, including its nonterminals, see "The for Statement" on page 156.)

This is an example of the **continue** statement:

```
while( i-- > 0 )
{
    x = f( i );
    if( x == 1 )
        continue;
    y += x * x;
}
```

In this example, the statement body is executed while `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 the loop's test.

5.5 The do-while Statement

The **do-while** statement lets you repeat a statement or compound statement until a specified expression becomes false.

Syntax

iteration-statement :
  do *statement* **while (** *expression* **)**

The *expression* in a **do-while** statement is evaluated after the body of the loop is executed. Therefore, the body of the loop is always executed at least once.

The *expression* must have arithmetic or pointer type. Execution proceeds as follows:

1. The statement body is executed.
2. Next, *expression* is evaluated. If *expression* is false, the **do-while** statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the process is repeated, beginning with step 1.

The **do-while** statement may also terminate when a **break**, **goto**, or **return** statement is executed within the statement body.

This is an example of the **do-while** statement:

```
do
{
    y = f( x );
    x--;
} while( x > 0 );
```

In this **do-while** 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-while** statement terminates when `x` becomes 0 or negative. The body of the loop is executed at least once.

5.6 The Expression Statement

When an expression statement is executed, the expression is evaluated according to the rules outlined in Chapter 4, "Expressions and Assignments."

Syntax

expression-statement :
  expression $_{\text{opt}}$ **;**

All side effects from the expression evaluation are completed before the next statement is executed. An empty expression statement is called a null statement. See "The Null Statement" on page 159.

These examples demonstrate expression statements.

```
x = ( y + 3 );              /* x is assigned the value of y + 3  */
x++;                        /* x is incremented                  */
x = y = 0;                  /* Both x and y are initialized to 0 */
proc( arg1, arg2 );         /* Function call returning void      */
y = z = ( f( x ) + 3 );     /* A function-call expression        */
```

In the last statement, the function-call expression, the value of the expression, which includes any value returned by the function, is increased by 3 and then assigned to both the variables `y` and `z`.

5.7 The for Statement

The **for** statement lets you repeat a statement or compound statement a specified number of times. The body of a **for** statement is executed zero or more times until an optional condition becomes false. You can use optional expressions within the **for** statement to initialize and change values during the **for** statement's execution.

Syntax

iteration-statement :
  for (*init-expression* $_{\text{opt}}$; *cond-expression* $_{\text{opt}}$; *loop-expression* $_{\text{opt}}$) *statement*

Execution of a **for** statement proceeds as follows:

1. The *init-expression,* if any, is evaluated. This specifies the initialization for the loop. There is no restriction on the type of *init-expression.*
2. The *cond-expression,* if any, is evaluated. This expression must have arithmetic or pointer type. It is evaluated before each iteration. Three results are possible:
   a. If *cond-expression* is true (nonzero), *statement* is executed; then *loop-expression,* if any, is evaluated. The *loop-expression* is evaluated after each iteration. There is no restriction on its type. Side effects will execute in order. The process then begins again with the evaluation of *cond-expression.*
   b. If *cond-expression* is omitted, *cond-expression* is considered true, and execution proceeds exactly as described for case a. A **for** statement without a *cond-expression* argument 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 *cond-expression* is false (0), 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. A **continue** statement in a **for** loop causes *loop-expression* to be evaluated. When a **break** statement is executed inside a **for** loop, *loop-expression* is not evaluted or executed. This statement

```
for( ;; );
```

is the customary way to produce an infinite loop which can only be exited with a **break**, **goto**, or **return** statement.

This example illustrates the **for** statement:

```
for ( i = space = tab = 0; i < MAX; i++ )
{
    if( line[i] == ' ' )
        space++;
    if( line[i] == '\t' )
    {
        tab++;
        line[i] = ' ';
    }
}
```

This example counts space (`' '`) 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 with 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`.

5.8 The goto and Labeled Statements

The **goto** statement transfers control to a label. The given label must reside in the same function and can appear before only one statement in the same function.

Syntax

statement :
  labeled-statement
  jump-statement

jump-statement :
  goto *identifier* **;**

labeled-statement :
  identifier **:** *statement*

A statement label is meaningful only to a **goto** statement; in any other context, a labeled statement is executed without regard to the label.

A *jump-statement* must reside in the same function and can appear before only one statement in the same function. The set of *identifier* names following a **goto** has its own name space so the names do not interfere with other identifiers. Labels cannot be redeclared. See "Name Spaces" on page 39 for information about name spaces.

It is good programming style to use the **break**, **continue**, and **return** statement in preference to **goto** whenever possible. Since the **break** statement only exits from one level of the loop, a **goto** may be necessary for exiting a loop from within a deeply nested loop.

This example demonstrates the **goto** statement:

```
void main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 3; j++ )
        {
            printf( " Inner loop executing. j = %d\n", j );
            if ( i == 5 )
                goto stop;
        }
    }
    /* This message does not print: */
    printf( "Loop exited. i = %d\n", i );
    stop: printf( "Jumped to stop. i = %d\n", i );
}
```

In this example, a **goto** statement transfers control to the point labeled `stop` when `i` equals 5.

5.9 The if Statement

The **if** statement controls conditional branching. The body of an **if** statement is executed if the value of the expression is nonzero. The syntax for the **if** statement has two forms.

Syntax

selection-statement:
  if (*expression*) *statement*
  if (*expression*) *statement1* **else** *statement2*

In both forms of the **if** statement, the expressions, which may have any value except a structure, are evaluated, including all side effects.

In the first form of the syntax, if *expression* is true (nonzero), *statement* is executed. If *expression* is false, *statement* is ignored. In the second form of syntax, which uses **else**, *statement2* is executed if *expression* is false. With both forms, control then passes from the **if** statement to the next statement in the program unless one of the statements contains a **break**, **continue**, or **goto**.

The following are examples of the **if** statement:

```
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.

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** that lacks an **else**.

```
if( i > 0 )            /* Without braces */
    if( j > i )
        x = j;
    else
        x = i;
```

The **else** clause is associated with the inner **if** statement in this example. If `i` is less than or equal to 0, no value is assigned to `x`.

```
if( i > 0 )
{                       /* With braces */
    if( j > i )
        x = j;
}
else
    x = i;
```

The braces surrounding the inner **if** statement in this example make the **else** clause part of the outer **if** statement. If `i` is less than or equal to 0, `i` is assigned to `x`.

5.10 The Null Statement

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. The correct way to code a null statement is:

Syntax ;

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.

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.

This example illustrates the null statement:

```
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.

5.11 The return Statement

The **return** statement terminates the execution of a function and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. A **return** statement can also return a value to the calling function. For more information see "Return Type" on page 177.

Syntax

jump-statement:
  return *expression* $_{\text{opt}}$ **;**

The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined. The expression, if present, is converted to the type returned by the function. If the function was declared with return type **void**, a **return** statement containing an expression is not legal.

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. In this case, the return value of the called function is undefined. If a return value is not required, declare the function to have **void** return type; otherwise the default return type is **int**.

Many programmers use parentheses to enclose the *expression* argument of the **return** statement. However, C does not require the parentheses.

This example demonstrates the **return** statement:

```
void draw( int I, long L );
long sq( int s );
```

```
int main()
{
    long y;
    int x;

    y = sq( x );
    draw( x, y );
    return();
}

long sq( int s )
{
    return ( s * s );
}

void draw( int I, long L )
{
    /* Statements defining the draw function here */
    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.

5.12 The switch Statement

The **switch** and **case** statements help control complex conditional and branching operations. The **switch** statement transfers control to a statement within its body.

Syntax

selection-statement :
  switch (*expression*) *statement*

labeled-statement :
  case *constant-expression* **:** *statement*
  default : *statement*

Control passes to the statement whose **case** *constant-expression* matches the value of **switch** (*expression*). The **switch** statement can include any number of **case** instances, but no two case constants within the same **switch** statement can have the same value. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a **break** statement transfers control out of the body.

Use of the **switch** statement usually looks something like this:

switch (*expression*)
{
    declarations
    .
    .
    .
    case *constant-expression* **:**

        statements executed if the expression equals the
        value of this constant-expression
        .
        .
        .
        break;
    default :
        statements executed if expression does not equal
        any case constant-expression
}

You can use the **break** statement to end processing of a particular case within the **switch** statement and to branch to the end of the **switch** statement. Without **break**, the program continues to the next case, executing the statements until a **break** or the end of the statement is reached. In some situations, this continuation may be desirable.

The **default** statement is executed if no **case** *constant-expression* is equal to the value of **switch** (*expression*). If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body are executed. There can be at most one **default** statement. The **default** statement need not come at the end; it can appear anywhere in the body of the **switch** statement. In fact it is often more efficient if it appears at the beginning of the **switch** statement. A **case** or **default** label can only appear inside a **switch** statement.

The type of **switch** *expression* and **case** *constant-expression* must be integral. The value of each **case** *constant-expression* must be unique within the statement body.

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. Switch statements can be nested. Any static variables are initialized before executing into any **switch** statements.

Note Declarations can 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.

The following examples illustrate **switch** statements:

```
switch( c )
{
    case 'A':
        capa++;
    case 'a':
        lettera++;
    default :
        total++;
}
```

All three statements of the **switch** body in this example are executed if `c` is equal to `'A'` since a **break** statement does not appear before the following case. 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'`.

```
switch( i )
{
    case -1:
        n++;
        break;
    case 0 :
        z++;
        break;
    case 1 :
        p++;
        break;
}
```

In this example, 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.

A single 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);
```

In this example, if *constant-expression* equals any letter between `'a'` and `'f'`, the `hexcvt` function is called.

Microsoft Specific

Microsoft C does not limit the number of case values in a **switch** statement. The number is limited only by the available memory. ANSI C requires at least 257 case labels be allowed in a **switch** statement.

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za command-line option to disable these extensions.♦

5.13 The while Statement

The **while** statement lets you repeat a statement until a specified expression becomes false.

Syntax

iteration-statement :
  while (*expression*) *statement*

The *expression* must have arithmetic or pointer type. Execution proceeds as follows:

1. The *expression* is evaluated.
2. If *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. Use the **continue** statement to terminate an iteration without exiting the **while** loop. The **continue** statement passes control to the next iteration of the **while** statement.

This is an example of the **while** statement:

```
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.

Functions

Chapter 6

The function is the fundamental modular unit in C. A function is usually designed to perform a specific task, and its name often reflects that task. A function contains declarations and statements. This chapter describes how to declare, define, and call C functions. Other topics discussed are:

- Function attributes such as **__near** and **__far**
- Calling conventions such as **__cdecl**, **__pascal**, and **__fortran**
- Export, inline, and interrupt functions
- Storage classes for functions
- Return types
- Function arguments and parameters

Function declarations are also discussed in Chapter 3, "Declarations and Types."

6.1 Overview

Functions must have a definition and should have a declaration, although a definition can serve as a declaration if the declaration appears before the function is called. The function definition includes the function body—the code that executes when the function is called.

A function declaration establishes the name, return type, and attributes of a function that is defined elsewhere in the program. A function declaration must precede the call to the function. This is why the header files containing the declarations for the run-time functions are included in your code prior to a call to a run-time function. If the declaration has information about the types and number of parameters, the declaration is a prototype. See "Function Prototypes" on page 181 for more information.

The compiler uses the prototype to compare the types of arguments in subsequent calls to the function with the function's parameters and to convert the types of the arguments to the types of the parameters whenever necessary.

A function call passes execution control from the calling function to the called function. The 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.

Obsolete Forms of Function Declarations and Definitions

The old-style function declarations and definitions use slightly different rules for declaring parameters than the syntax recommended by the ANSI standard. First, the old-style declarations don't have a parameter list. Second, in the function definition, the parameters are listed, but their types are not declared in the parameter list. The type declarations precede the compound statement constituting the function body. The old-style syntax is obsolete and should not be used in new code. Code using the old-style syntax is still supported, however. This example illustrates the obsolete forms of declarations and definitions:

```
double old_style();          /* Obsolete function declaration */

double alt_style( a , real ) /* Obsolete function definition */
    double *real;
    int a;
{
    return ( *real + a ) ;
}
```

Functions returning an integer or pointer with the same size as an **int** are not required to have a declaration although the declaration is recommended.

The next section shows the syntax for function definitions, including the old-style syntax. The nonterminal for the list of parameters in the old-style syntax is *identifier-list.*

6.2 Function Definitions

A function definition specifies the name of the function, the types and number of parameters it expects to receive, and its return type. A function definition also includes a function body with the declarations of its local variables, and the statements that determine what the function does.

Syntax

translation-unit:
  external-declaration
  translation-unit external-declaration

external-declaration : /* Allowed only at external (file) scope */
function-definition
declaration

function-definition : /* Declarator here is the function declarator */
declaration-specifiers $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

Prototype parameters are:

declaration-specifiers :
storage-class-specifier declaration-specifiers $_{\text{opt}}$
type-specifier declaration-specifiers $_{\text{opt}}$
type-qualifier declaration-specifiers $_{\text{opt}}$
attributes $_{\text{opt}}$ *declaration-specifiers* $_{\text{opt}}$ /* Microsoft-specific */

declaration-list :
declaration
declaration-list declaration

declarator :
pointer $_{\text{opt}}$ *direct-declarator*

direct-declarator : /* A function declarator */
direct-declarator (*parameter-type-list*) /* New-style declarator */
direct-declarator (*identifier-list* $_{\text{opt}}$) /* Obsolete-style declarator */

The parameter list in a definition uses this syntax:

parameter-type-list : /* The parameter list */
parameter-list
parameter-list , **...**

parameter-list :
parameter-declaration
parameter-list , *parameter-declaration*

parameter-declaration :
declaration-specifiers declarator
declaration-specifiers abstract-declarator $_{\text{opt}}$

The parameter list in an old-style function definition uses this syntax:

identifier-list: /* Used in obsolete-style function definitions and declarations */
identifier
identifier-list , *identifier*

The syntax for the function body is

compound-statement : /* The function body */
{ *declaration-list* $_{\text{opt}}$ *statement-list* $_{\text{opt}}$ }

The only storage-class specifiers that can modify a function declaration are **extern** and **static**. The **extern** specifier signifies that the function can be referenced from other files; that is, the function name is exported to the linker. The **static** specifier signifies that the function cannot be referenced from other files; that is, the name is not exported by the linker. If no storage class appears in a function definition, **extern** is assumed. In any case, the function is always visible from the definition point to the end of the file.

The optional *declaration-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. The *attributes* nonterminal is a Microsoft-specific feature defined in the next section, "Function Attributes."

The *direct-declarator* (in the *declarator* syntax) specifies the name of the function being defined and the identifiers of its parameters. If the *direct-declarator* includes a *parameter-type-list*, the list specifies the types of all the parameters. Such a declarator also serves as a function prototype for later calls to the function.

A *declaration* in the *declaration-list* in function definitions cannot contain a *storage-class-specifier* other than **register**. The *type-specifier* in the *declaration-specifiers* syntax can be omitted only if the **register** storage class is specified for a value of **int** type.

The *compound-statement* is the function body containing local variable declarations, references to externally declared items, and statements.

The sections "Function Attributes" on page 168 through "Function Body" on page 181 describe the components of the function definition in detail.

Function Attributes

The optional *attributes* nonterminal allows you to override default addressing modes by specifying a different memory model, or to select a calling convention on a per-function basis to override the defaults. You can also specify functions as **__fastcall**, **__export**, **__inline**, **__based**, inline assembler, or **__interrupt**, and can specify register-handling with **__loadds** and **__saveregs**.

Specifying Function Addressing (Microsoft Specific)

By using the special keywords **__near**, **__far**, **__huge**, and **__based**, you can override the addressing specified by the compile-time memory models. The following list summarizes use of these keywords in function *attributes*.

__near
Functions are assumed to be in the default code segment (_TEXT). The function is referenced with 16-bit addresses (pointers to functions are 16 bits) and can be called only by functions in the same code segment. Functions declared as **__near** can be allocated in other segments by declaring them as **__based** or using the /NT compilation option.

32-Bit Specific

The use of the **__near** keyword is not allowed for 32-bit targets.♦

__far
Functions are not assumed to be in the current code segment. Far objects are referenced with 32-bit addresses (pointers to functions are 32 bits) and can be called with a far call by functions anywhere in memory. Functions in the same compilation unit reside in the same segment unless the **alloc_text** pragma is used, or unless the function is declared as based.

32-Bit Specific

The use of the **__far** keyword is not allowed for 32-bit targets.♦

__based
Specifies that a function resides in a specified segment. More information on **__based** appears later in this section.

__huge
The **__huge** keyword is not applicable to functions.

This example uses the **__far** keyword:

```
void __far handler( unsigned doserr, unsigned __far *bdr );
```

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za command-line option to disable these extensions.

Specifying Calling Conventions (Microsoft Specific)

The special keywords discussed in this section allow you to directly specify the calling convention for any function. This list summarizes these Microsoft-specific keywords.

__cdecl
Specifies that the associated function is to be called using the normal C calling convention (arguments are pushed from right to left, the calling function adjusts the stack, and no case translation takes place). This modifier is placed before

the function name, and can appear before or after the __**near** and __**far** modifiers. The /Gd command-line option forces the __**cdecl** calling convention.

__**pascal,** __**fortran**
Specifies that the associated function is to be called using the Pascal or FORTRAN calling convention (arguments are pushed from left to right, the called function adjusts the stack, and identifier names are translated to uppercase). These modifiers are placed before the function name, and can appear before or after the __**near** and __**far** modifiers. The __**fortran** and __**pascal** modifiers are synonyms. The /Gc command-line option forces the __**pascal** or __**fortran** calling convention.

32-Bit Specific

The __**fortran** and __**pascal** keywords are not supported for 32-bit targets. ♦

__**fastcall**
Specifies that the function uses a calling convention that passes arguments in registers rather than on the stack, resulting in faster code for small functions. Arguments are passed from left to right, the called function adjusts the stack, and no case translation takes place. Using /Gr on the command line causes each function in the module to compile as fastcall unless the function is declared with a conflicting attribute, or the name of the function is `main`. See the next section, "Fastcall Functions," for more information.

__**stdcall**
Specifies that the arguments of the designated function are pushed from right to left, that an underscore is prepended to the name, and *@###* is appended, where *###* is the number of bytes in the parameters of the function. This appended "name decoration" allows link-time argument checking to be done. The called function does its own stack adjustment when it returns to the caller. Functions declared with __**stdcall** return values the same way as functions declared using __**cdecl**. If a __**stdcall** function has a variable number of arguments, it must have a prototype and it will be implemented as __**cdecl**. The /Gz command-line option specifies __**stdcall** for all functions that are not explicitly declared with a different calling convention.

32-Bit Specific

The __**stdcall** calling convention is only available to 32-bit compilations.♦

This example declaration specifies the __**cdecl** calling convention:

```
int __cdecl compare( unsigned *key );
```

Fastcall Functions (Microsoft Specific)

A __**fastcall** function in programs for 16-bit targets receives up to three 16-bit arguments passed in registers rather than on the stack. The choice of registers for the __**fastcall** calling convention depends on the type of arguments:

Type	Register Candidates
char / unsigned char	AL, DL, BL
int / unsigned int	AX, DX, BX
long / unsigned long	DX:AX
near pointer	BX, AX, DX
far or huge pointer	Passed on the stack

Arguments are allocated to suitable registers if available and are pushed onto the stack otherwise. Structures, unions, and all floating-point types are always pushed onto the stack. Return values of four bytes or smaller, including structures and unions, are placed in the registers as follows:

Size	Register
1 byte	AL
2 bytes	AX
4 bytes	DX:AX

The implementation of fastcall functions is processor-dependent. On the 8086 and 80286, floating-point values are returned on the floating-point stack. To return structures or unions larger than four bytes, the calling program pushes a hidden last parameter, which is a near pointer to a buffer in which the value is to be returned. A far pointer to the hidden-parameter must be returned in DX:AX.

For the 80386 and 80486 processors, the first two arguments that have integer or pointer types are passed in the ECX and EDX registers, regardless of size. The rest of the arguments are passed on the stack from left to right. Regardless of length, structure returns are handled with a hidden parameter.

The treatment of character arguments depends further on prototypes. If there is no prototype, the argument is promoted to **short** and the rules for **short** integers apply. Only if the argument is prototyped to type **char** do the character rules apply.

The **__fastcall** calling convention cannot be used with functions having variable-length parameter lists, or functions having any of the following attributes: **__cdecl**, **__export**, **__fortran**, **__interrupt**, **__pascal**, **__saveregs**.

Note Microsoft does not guarantee its implementation of the fastcall calling convention between releases.

Export Functions (Microsoft Specific)

The **__export** keyword allows you to specify that a function is to be exported from a dynamic-link library (DLL) to Windows. Use this form to specify an exported function:

__export *declarator*

The *declarator* is the name of the exported function. When a function is declared as **__export**, the compiler places information in the object file to show that the function is exported from a DLL. Functions, operators, and data can be declared as **__export**.

The main use for **__export** is to export symbols that reside in a DLL. You also may need to export event-handler functions for Microsoft Windows programs, or for PWB extensions.

The **__export** keyword does not eliminate the need for a module-definition (.DEF) file when building a DLL. If you declare a symbol as **__export** and no .DEF file entry exists for that symbol, the linker assumes that the symbol has the following characteristics:

- No input/output (I/O) privilege
- Shared data
- Not resident
- No alias name

If these default characteristics are satisfactory, the symbol does not require an entry in a module-definition file. Otherwise, you must create an **EXPORTS** entry in the module-definition file for the symbol to specify these characteristics.

The **__export** keyword also causes the compiler to enter the size, in words, of the function's parameters into the export record of the object module. This size information corresponds to the *pwords* field of an **EXPORTS** statement that is in a module-definition file. You cannot override the size information in the export record with an **EXPORTS** entry in the .DEF file.

If you have an **EXPORTS** entry for a function, the *pwords* field in the .DEF file should be set either to 0 (which tells the linker to use the value given by the compiler) or to the same value given by the compiler. The *pwords* field is ignored unless you also request I/O privilege. For more information on creating .DEF files and import libraries, see Chapters 16 and 22, respectively, in the *Environment and Tools* manual.

The following statement declares `funcsample` as a far Pascal function that takes a single argument of any pointer type and does not return a value.

```
void __export __far __pascal funcsample( void *s );
```

The presence of **__export** causes the function to be exported. A .DEF file is still required for the program.

Inline Functions (Microsoft Specific)

The **__inline** keyword tells the compiler that it can substitute the code within the function definition for every instance of a function call. Substitution occurs at the discretion of the compiler. Use this form to specify an inline function:

__inline *type* $_{\text{opt}}$ *function_definition*;

The use of inline functions generates faster code and can sometimes generate smaller code than the equivalent function call generates for the following reasons:

- It saves the time required to execute function calls.
- Small inline functions, perhaps three lines or less, create less code than the equivalent function call because the compiler doesn't generate code to handle arguments and a return value.
- Functions generated inline are subject to code optimizations not available to normal functions because the compiler does not perform interprocedural optimizations.

Functions using **__inline** should not be confused with inline assembler code. See "Inline Assembler" on page 174 for more information.

Function Addressing Using __based (Microsoft Specific)

If a function is to be allocated in a given segment, it can be declared as based on a segment constant. The base for a function can be specified as

__based(__segname(*string-literal* **))**

A function declared as **__based** resides in the code segment named by *string-literal*. The built-in function **__segname** accepts a string enclosed in quotation marks and returns a value of type **__segment**. The **__segment** type specifies the segment in which the based function resides. You can use the **__near** or **__far** keyword with the **__based** keyword when declaring a function.

In programs that use overlays, you can reduce swapping by using **__based** to group functions that frequently call one another. You also can use **__based** to ensure that near functions reside in the same segment as the functions that call them.

Declaring functions as **__based** replaces the **alloc_text** pragma. However, the **alloc_text** pragma is retained for backward compatibility.

The base for a function based in a segment can also be specified as

(__based)__self

The **__self** function ensures that the function's location is the segment in which the pointer itself is stored. Such pointers can save space in a linked list or tree if the entire data structure fits in a single segment.

Inline Assembler (Microsoft Specific)

The inline assembler lets you embed assembly-language instructions directly in your C source programs without extra assembly and link steps. The inline assembler is built into the compiler—you don't need a separate assembler such as the Microsoft Macro Assembler (MASM).

Because the inline assembler doesn't require separate assembly and link steps, it is more convenient than a separate assembler. Inline assembly code can use any C variable or function name that is in scope, so it is easy to integrate it with your program's C code. And because the assembly code can be mixed with C statements, it can do tasks that are cumbersome or impossible in C alone.

The **__asm** keyword invokes the inline assembler and can appear wherever a C statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces. The term "**__asm** block" here refers to any instruction or group of instructions, whether or not in braces.

Below is a simple **__asm** block enclosed in braces. (The code prints the "beep" character, ASCII 7.)

```
__asm
{
   mov ah, 2
   mov dl, 7
   int 21h
}
```

Alternatively, you can put **__asm** in front of each assembly instruction:

```
__asm mov ah, 2
__asm mov dl, 7
__asm int 21h
```

Since the **__asm** keyword is a statement separator, you can also put assembly instructions on the same line:

```
__asm mov ah, 2   __asm mov dl, 7   __asm int 21h
```

For more information, see Chapter 6 of the *Programming Techniques* manual.

Interrupt Functions (Microsoft Specific)

The __**interrupt** keyword specifies that the function is an interrupt handler. The compiler generates appropriate entry and exit sequences for the handling function, including saving and restoring all registers and executing an **IRET** instruction to return. Use this form to specify an interrupt function:

__**interrupt** *declarator*

where *declarator* is the name of the function to be called. An interrupt function must be __**far**. If you are compiling with the small (default) or compact memory model, you must explicitly declare the function with the __**far** attribute. An interrupt function cannot be declared as an inline function.

Interrupt functions must observe the C calling convention. If you use the /Gc compiler option (forcing the __**pascal** or __**fortran** calling convention) or the /Gr compiler option (forcing the __**fastcall** calling convention), you must explicitly declare your interrupt-handling function with the __**cdecl** attribute.

You cannot declare an interrupt function with both the __**interrupt** attribute and the __**saveregs** attribute or the __**fastcall** calling convention.

This example statement declares a function pointer that can be used to point to an interrupt handler:

```
void ( __interrupt __far *oldtime ) ( void );
```

The __**interrupt** keyword is implemented for 32-bit targets. See Help for more information on writing interrupt functions.

Using __loadds and __saveregs (Microsoft Specific)

The __**loadds** keyword causes the data-segment (DS) register to be loaded with a specified segment value upon entering the specified function. The previous DS value is restored when the function terminates. Use this form for __**loadds**:

__**loadds** *declarator*

The *declarator* specifies the function name of a function that must load DS as part of its entry sequence. Loading the DS register is essential for Windows callback functions and Windows entry points. The __**loadds** keyword does not imply any change in calling convention. It can be specified with any calling-convention modifier.

The compiler uses the segment name specified by the /ND (name-data-segment) option, or, if no segment has been specified, the default group, DGROUP. The __**loadds** attribute has the same effect as the /Au option, but on a function-by-function basis.

The **__loadds** keyword does not imply any change in the calling convention. It can be specified with any calling convention attribute that is supported for 16-bit targets.

The **__saveregs** keyword causes the compiler to generate code that saves all CPU registers when entering a function and also the code that restores the registers on exit. Note that **__saveregs** does not restore registers used for a return value (the AX register, or AX and DX). The form for using **__saveregs** is:

__saveregs *declarator*

The *declarator* specifies the function name whose entry sequence must save the values of all registers. The **__saveregs** keyword is useful when the register conventions of the caller are unknown. It is illegal to declare a function with both the **__saveregs** and **__interrupt** attributes.

Storage Class

The storage-class specifier in a function definition gives the function either **extern** or **static** storage class.

Syntax

function-definition :
  declaration-specifiers $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

declaration-specifiers :
  storage-class-specifier declaration-specifiers $_{\text{opt}}$
  type-specifier declaration-specifiers $_{\text{opt}}$
  type-qualifier declaration-specifiers $_{\text{opt}}$

storage-class-specifier :   /* For function definitions */
  extern
  static

If a function definition does not include a *storage-class-specifier*, the storage class defaults to **extern**. You can explicitly declare a function as **extern**, but it is not required.

If the declaration of a function contains the *storage-class-specifier* **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage. If an identifier has file scope and no *storage-class-specifier*, the identifier has external linkage. External linkage means that each instance of the identifier denotes the same object or function. See "Understanding Lifetime, Scope, Visibility, and Linkage" on page 34 for more information about linkage and file scope.

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. If **static** storage class is desired, it must be declared on the first occurrence of a declaration (if any) of the function, and on the definition of the function.

Microsoft Specific

When the Microsoft extensions are enabled, a function originally declared without a storage class (or with **extern** storage class) is given **static** storage class if the function definition is in the same source file and if the definition explicitly specifies **static** storage class.♦

Return Type

The return type of a function establishes the size and type of the value returned by the function and corresponds to the type-specifier in the syntax below:

Syntax

function-definition :
  declaration-specifiers $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

declaration-specifiers :
  storage-class-specifier declaration-specifiers $_{\text{opt}}$
  type-specifier declaration-specifiers $_{\text{opt}}$
  type-qualifier declaration-specifiers $_{\text{opt}}$

type-specifier :
  void
  char
  short
  int
  long
  float
  double
  signed
  unsigned
  struct-or-union-specifier
  enum-specifier
  typedef-name

The *type-specifier* can specify any fundamental, structure, or union type. If you do not include *type-specifier*, the return type **int** is assumed.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. A function returns a value 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.

The following examples illustrate function return values.

```
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 );
}
```

This 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. In subsequent calls to the function, the compiler checks to make sure the argument types are `STUDENT`.

Note Efficiency would be enhanced by passing pointers to the structure, rather than the entire structure.

```
char *smallstr( char s1[], char s2[] )
{
    int i;

    i = 0;
    while( s1[i] != '\0' && s2[i] != '\0' )
        i++;
    if( s1[i] == '\0' )
        return ( s1 );
    else
        return ( s2 );
}
```

This example defines 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 first of the array elements and has its type; thus, the return type of the function is a pointer to type **char**.

Microsoft Specific

You need not declare functions with **int** return type before you call them, although prototypes are recommended so that correct type checking for arguments and return values is enabled. ♦

Parameters

Arguments are names of values passed to a function by a function call. Parameters are the values the function expects to receive. In a function prototype, the parentheses following the function name contain a complete list of the function's parameters and their types. Parameter declarations specify the types, sizes, and identifiers of values stored in the parameters.

Syntax

function-definition :
  declaration-specifiers opt *declarator declaration-list* opt *compound-statement*

declarator :
  pointer opt *direct-declarator*

direct-declarator : /* A function declarator */
  direct-declarator (*parameter-type-list*) /* New-style declarator */

parameter-type-list : /* A parameter list */
  parameter-list
  parameter-list **, ...**

parameter-list :
  parameter-declaration
  parameter-list **,** *parameter-declaration*

parameter-declaration :
  declaration-specifiers declarator
  declaration-specifiers abstract-declarator opt

The *parameter-type-list* is a sequence of parameter declarations separated by commas. The form of each parameter in a parameter list looks like this:

[[**register**]] *type-specifier* [[*declarator*]]

The identifiers of the parameters are used in the function body to refer to the values passed to the function. You can name the parameters in a prototype, but the names go out of scope at the end of the declaration. Therefore parameter names can be assigned the same way or differently in the function definition. These identifiers cannot be redefined in the outermost block of the function body, but they can be redefined in inner, nested blocks as though the parameter list were an enclosing block.

Each identifier in *parameter-type-list* must be preceded by its appropriate type specifier, as shown in this example:

```
void new( double x, double y, double z )
{
    /* Function body here */
}
```

If at least one parameter occurs in the 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. (See "Calls with a Variable Number of Arguments" on page 187.) However, a call to the function must have at least as many arguments as there are parameters before the last comma.

If no arguments are to be passed to the function, the list of parameters is replaced by the keyword **void**. This use of **void** is distinct from its use as a type specifier.

The order and type of parameters, including any use of the ellipsis notation, must be the same in all the function declarations (if any) and in the function definition. The types of the arguments after usual arithmetic conversions must be assignment-compatible with the types of the corresponding parameters. (See "Usual Arithmetic Conversions" on page 115 for information on arithmetic conversions.) Arguments following the ellipsis are not checked. A parameter can have any fundamental, structure, union, pointer, or array type.

The compiler performs the usual arithmetic conversions independently on each parameter and on each argument, if necessary. After conversion, no parameter is shorter than an **int**, and no parameter has **float** type unless the parameter type is explicitly specified as **float** in the prototype. This means, for example, that declaring a parameter as a **char** has the same effect as declaring it as an **int**.

When the **__near**, **__far**, and **__huge** keywords appear in the declaration, 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 "Assignment Conversions" on page 141 for more information on pointer conversions.

Microsoft Specific

Microsoft C/C++ version 7.0 allows you to mix complete parameter declarations (`int a`) and abstract declarators (`int`) in the same declaration. For example, the following declaration is legal in Microsoft C/C++ version 7.0 but not in Microsoft C version 6.0:

```
int add( int a, int );
```

To maintain compatibility with Microsoft C version 6.0, a Microsoft extension to the ANSI C standard allows a comma without trailing periods (**,**) at the end of the list of parameters to indicate a variable number of arguments. However, it is recommended that code be changed to incorporate the ellipsis notation.♦

Function Body

A "function body" is a compound statement containing the statements that specify what the function does. The syntax is:

Syntax

function-definition :
  declaration-specifiers $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

compound-statement :   /* The function body */
  { *declaration-list* $_{\text{opt}}$ *statement-list* $_{\text{opt}}$ }

Variables declared in a function body, "local variables," 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 *compound-statement* and continues 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.

6.3 Function Prototypes

A function declaration precedes the function definition and specifies the name, return type, storage class, and other attributes of a function. To be a prototype, the function declaration must also establish types and identifiers for the function's arguments.

Syntax

declaration :
  declaration-specifiers init-declarator-list $_{opt}$ **;**

declaration-specifiers :
  storage-class-specifier declaration-specifiers $_{opt}$
  type-specifier declaration-specifiers $_{opt}$
  type-qualifier declaration-specifiers $_{opt}$

init-declarator-list :
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator :
  declarator
  declarator = *initializer*

declarator:
  pointer $_{opt}$ *direct-declarator*

direct-declarator : /* A function declarator */
  direct-declarator (*parameter-type-list*) /* New-style declarator */
  direct-declarator (*identifier-list* $_{opt}$) /* Obsolete-style declarator */

The prototype has the same form as the function definition, except that it is terminated by a semicolon immediately following the closing parenthesis and therefore has no body. In either case, the return type must agree with the return type specified in the function definition.

Function prototypes have the following important uses:

- They establish the return type for functions that return types other than **int**. Although functions that return **int** values do not require prototypes, prototypes are recommended.
- Without complete prototypes, standard conversions are made, but no attempt is made to check the type or number of arguments with the number of parameters.
- Prototypes are used to initialize pointers to functions before those functions are defined.
- The parameter list is used for checking the correspondence of arguments in the function call with the parameters in the function definition.

The converted type of each parameter determines the interpretation of the arguments that the function call places on the stack. A type mismatch between an argument and a parameter may cause the arguments on the stack to be misinterpreted. For example, on a 16-bit computer, if a 16-bit pointer is passed as an argument, then declared as a **long** parameter, the first 32 bits on the stack are interpreted as a **long** parameter. This error creates problems not only with the **long** parameter, but with any parameters that follow it. You can detect errors of this kind by declaring complete function prototypes for all functions.

A prototype establishes the attributes of a function so that calls to the function that precede its definition (or occur in other source files) can be checked for argument-type and return-type mismatches. For example, if you specify the **static** storage-class specifier in a prototype, you must also specify the **static** storage class in the function definition.

The prototype can include both the type of, and an identifier for, each expression that is passed as an 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 is variable, or that no arguments are passed. Without such a list, mismatches may not be revealed, so the compiler cannot generate diagnostic messages concerning them. For more information on type checking, see "Arguments" on page 185.

6.4 Function Calls

A function call is an expression that passes control and arguments (if any) to a function and has the form

expression (*expression-list* $_{\text{opt}}$)

where *expression* is a function name or evaluates to a function address and *expression-list* is a list of expressions (separated by commas). The values of these latter expressions are the arguments passed to the function. If the function does not return a value, then you declare it to be a function that returns **void**.

If a declaration exists before the function call, but no information is given concerning the parameters, any undeclared arguments simply undergo the usual arithmetic conversions.

Note The expressions in the function argument list can be evaluated in any order, so arguments whose values may be changed 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. (Note that the order in which arguments are pushed on the stack is a separate matter.) See "Sequence Points" on page 110 for more information.

The only requirement in any 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.

This example illustrates function calls called from a **switch** statement:

```
main()
{
    /* Function prototypes */

    long lift( int ), step( int ), drop( int );
    void work( int number, long (*function)(int i) );

    int select, count;
    .
    .
    .
    select = 1;
    switch( select )
    {
        case 1: work( count, lift );
                break;

        case 2: work( count, step );
                break;

        case 3: work( count, drop );
                /* Fall through to next case */
        default:
                break;
    }
}

/* Function definition */

void work( int number, long (*function)(int i) )
{
    int i;
    long j;

    for( i = j = 0; i < number; i++ )
            j += ( *function )( i );
}
```

In this example, the function call in `main`,

```
work( count, lift );
```

passes an integer variable, `count`, 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 for `work` is given at the beginning of the `main` function.

The 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 from inside the **for** loop by using the following function call:

```
( *function )( i );
```

One argument, `i`, is passed to the called function.

Arguments

The arguments in a function call have this form:

expression (*expression-list* $_{\text{opt}}$) /* Function call */

In a function call, *expression-list* is a list of expressions (separated by commas). The values of these latter expressions are the arguments passed to the function. If the function takes no arguments, *expression-list* should contain the keyword **void**.

An argument can be any value with fundamental, structure, union, or pointer type. All arguments are passed by value. This means a copy of the argument is assigned to the corresponding parameter. The function does not know the actual memory location of the argument passed. The function uses this copy without affecting the variable from which it was originally derived.

Although you cannot pass arrays or functions as arguments, you can pass pointers to these items. 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.

Compilers differ in the order they evaluate arguments. However, the arguments and any side effects are completely evaluated before the function is entered. See "Side Effects" on page 110 for information on side effects.

The *expression-list* in a function call is evaluated and the usual arithmetic conversions are performed on each argument in the function call. If a prototype is available, the resulting argument type is compared to the prototype's corresponding parameter. If they do not match, either a conversion is performed, or a diagnostic message is issued. The parameters also undergo the usual arithmetic conversions.

If the **__near**, **__far**, and **__huge** keywords are used, conversions on pointer arguments may also be performed. See "Assignment Conversions" on page 141 for more information.

The number of expressions in *expression-list* must match the number of parameters, unless the function's prototype 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 parameters and converts them, if necessary, as described above. See the following section, "Calls with a Variable Number of Arguments," for more information.

If the prototype's parameter list contains only the keyword **void**, the compiler expects zero arguments in the function call and zero parameters in the definition. A diagnostic message is issued if it finds any arguments.

This example uses pointers as arguments:

```
main()
{
    /* Function prototype */

    void swap( int *num1, int *num2 );
    int x, y;
    .
    .
    .
    swap( &x, &y );  /* Function call */
}

/* 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 **int** values. The parameters `num1` and `num2` in the prototype-style definition are also declared as pointers to **int** type values. 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 of `swap` includes argument types for each parameter. The identifiers within the parentheses of the prototype and definition can be the same or different. What is important is that the types of the arguments match those of the parameter lists in both the prototype and the definition.

Calls with a Variable Number of Arguments

A partial parameter list can be terminated by the ellipsis notation, a comma followed by three periods (**, ...**), to indicate that there may be more arguments passed to the function, but no more information is given about them. Type checking is not performed on such arguments. At least one parameter must precede the ellipsis notation and the ellipsis notation must be the last token in the parameter list. Without the ellipsis notation, the behavior of a function is undefined if it receives parameters in addition to those declared in the parameter list.

To call a function with a variable number of arguments, simply specify any number of arguments in the function call. An example is the **printf** function from the C run-time library. The function call must include one argument for each type name declared in the parameter list or the list of argument types.

All the arguments specified in the function call are placed on the stack unless the **__fastcall** calling convention is specified. The number of parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the parameters. You are responsible for retrieving any additional arguments from the stack and for determining how many arguments are present. The STDARGS.H file contains ANSI-style macros for accessing arguments of functions which take a variable number of arguments. The XENIX®-style macros in VARARGS.H are also still supported.

This sample declaration is for a function that calls a variable number of arguments:

```
int average( int first, ...);
```

Microsoft Specific

To maintain compatibility with Microsoft C version 6.0, a Microsoft extension to the ANSI C standard allows a comma without trailing periods (**,**) at the end of the list of parameters to indicate a variable number of arguments.♦

Recursive Functions

Any function in a C program can be called recursively; that is, it can call itself. The number of recursive calls is limited to the size of the stack. See Chapter 14 in the *Environment and Tools* manual for information about linker options that set stack size. Each time the function is called, new storage is allocated for the 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.

This example illustrates recursive calls:

```
int factorial( int num );      /* Function prototype */

void main()
{
    int result, number;
    .
    .
    .
    result = factorial( number );
}

int factorial( int num )      /* Function definition */
{
    .
    .
    .
    if( ( num > 0 ) || ( num <= 10 ) )
        return( num * factorial( num - 1 ) );
}
```

Chapter 7

Preprocessor Directives and Pragmas

A “preprocessor directive” is an instruction to the C preprocessor. Preprocessing takes place during the first phase of compilation. This chapter describes preprocessing and explains the “phases of translation,” which are the steps of the process by which source files (or translation units) are translated into executable files.

This chapter also discusses

- Macros
- The **#define** and **#undef** directives
- Preprocessing operators
- Include files (also known as header files)
- Conditional compilation
- Line control directives
- Error directives
- Pragma directives

A “pragma” is an instruction to the C compiler. Pragmas are allowed by ANSI as a way to implement vendor-specific behavior in the compiler.

7.1 Preprocessing

The C preprocessor manipulates the content of a source file as the first phase of compilation. The preprocessor does not parse the source text, but it does break it up into tokens for the purpose of locating macro calls. Although the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling.

Preprocessor directives are typically used to facilitate portability and improve program structure. 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. Preprocessor statements are recognized and carried out before macro expansion.

Preprocessor statements use the same character set as source file statements with the exception that escape sequences are not supported. (See "Character Constants" on page 16 for information on the execution character set.)

The C preprocessor recognizes the following directives:

#define	**#elif**	**#else**	**#endif**
#if	**#ifdef**	**#error**	**#include**
#line	**#undef**	**#ifndef**	**#pragma**

The number sign (#) must be the first nonwhite-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 (/* */) or be preceded by two consecutive forward slashes (//). Lines containing preprocessor directives can be continued by preceding the end-of-the-line marker with a backslash (\).

Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file.

7.2 Phases of Translation

A C program consists of one or more source files, each of which contains some of the text of the program. A source file, together with all of its "include files," which are files that are inserted at the location of the **#include** preprocessor directive, is called a "translation unit."

Source files are translated in a series of phases. Preprocessing treats a source file as a sequence of text lines. You can specify directives and macros to insert, delete, and alter source text. Once translated, the translation units can be kept either in separate object files or in object-code libraries. These separate translation units are then linked to form an executable program (.EXE or .COM file).

Functions in different translation units can pass values through:

- Calls to functions that have external linkage.
- Direct modification of identifiers that have external linkage.
- Direct modification of files.
- Interprocess communication (Windows only).

- Modification of environment variables.

The following list describes the phases in which the compiler translates files:

Character mapping
: Characters in the source file are mapped to the internal source representation. Trigraph sequences are converted to single-character internal representation in this phase. See page 8 for information on trigraphs.

Line splicing
: All lines ending in a backslash (\), immediately followed by a newline character, are joined with the next line in the source file, forming logical lines from the physical lines. A non-empty source file must end in a newline character that is not preceded by a backslash.

Tokenization
: The source file is broken into preprocessing tokens and white-space characters. Each comment in the source file is replaced with a space character. Newline characters are retained.

Preprocessing
: Preprocessing directives are executed and macros are expanded into the source file. The **#include** statement invokes the preprocessing steps starting with the preceding three translation processes on any included text.

Character set mapping
: All source-character-set members and escape sequences are converted to their equivalents in the execution character set. For Microsoft C/C++, both the source and the execution character sets are ASCII.

String concatenation
: All adjacent string literals and wide-string literals are concatenated. For example, `"String " "concatenation"` becomes `"String concatenation"`.

Translation
: All tokens are analyzed syntactically and semantically; these tokens are converted into object code.

The linker resolves all external references and creates an executable program by combining one or more separately processed translation units along with standard libraries.

7.3 Manifest Constants and Macros

The **#define** directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers defined with **#define** that represent constants are called "manifest constants" or "symbolic constants." Identifiers defined with **#define** that represent statements or expressions are called "macros."

This section discusses the **#define** directive for defining manifest constants and macros, the preprocessing operators you can use in macros, the **#undef** directive for removing macro and constant definitions, and the predefined macros provided in Microsoft C/C++ version 7.0.

Macros have their own name space. See page 39 for information on name spaces.

Macro Expansion

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 that they look and act like function calls. The next section gives the syntax for both kinds of macros. 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 as a function would. See the examples on page 193, "The #define Directive" for more information.

When the preprocessor encounters a macro, the macro is replaced by the macro body. If the macro accepts arguments, the arguments following the macro name are substituted for parameters in the macro body. The process of replacing a macro call with the processed copy of the body is called "expansion" of the macro call. Macros are not expanded recursively.

Microsoft Specific Macro expansions of up to 6K are permitted.♦

Preprocessing expands macros in all nondirective lines and in parts of some directives that are not skipped as part of a conditional compilation. Therefore, if a macro expands into something that looks like a preprocessor command, that command is not recognized by the preprocessor. (See page 202 for information on conditional compilation.)

Once you have defined a manifest constant or a macro, you cannot redefine it to a different value without first removing the original definition. However, you can redefine a manifest constant or a macro with exactly the same definition. This is useful if you have the same macro in several include files.

The **#undef** directive removes the definition of a manifest constant or a macro. Once you have removed the definition, you can redefine a manifest constant to a different value or a macro to a different statement without causing a compiler warning to be generated.

Using inline functions instead of function-like macros can be more reliable since parameters are type-checked, all expressions passed to a function are evaluated, and all side effects are complete prior to entry into the function. This is not necessarily true for macros. See information on **__inline** on page 173.

The #define Directive

You can use the **#define** directive to give a meaningful name to a constant or statement in your program. The ways to specify a manifest constant or a macro are given by this syntax:

Syntax

control-line :
 #define *identifier replacement-list new-line* /* Macro without parameters */
 #define *identifier* (*identifier-list* $_{\text{opt}}$ *new-line*) *replacement-list new-line*

The **#define** directive substitutes *replacement-list* for all subsequent occurrences of *identifier* in the source file. The *identifier* is replaced only when it forms a token. For instance, *identifier* is not replaced if it appears in a comment, within a string, or as part of a longer identifier.

The *replacement-list* argument consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate *replacement-list* from *identifier.* This initial white space is not considered part of the substituted text, nor is any white space following the last token of the text.

If an *identifier-list* appears after *identifier*, the **#define** directive replaces each occurrence of *identifier* (*identifier-list*) with a version of the *replacement-list* argument that has arguments substituted for parameters. The *identifier-list* is a list of parameters for a macro.

When a macro with parameters has been defined, subsequent textual instances followed by an *identifier-list* constitute a macro call. The arguments following an instance of *identifier* in the source file are matched to the corresponding parameters of *identifier-list.* Each parameter in *replacement-list* that is not preceded by a stringizing (#), charizing (#@), or token-pasting (##) operator, or followed by a token-pasting operator, is replaced by the corresponding argument. Any macros in the argument are expanded before the argument replaces the parameter. (The preprocessor operators are described on page 195.)

Parameter names appear in *replacement-list* to mark the places where actual values are substituted. A parameter name can appear more than once in *replacement-list*, and the names can appear in any order. The number of arguments in the call must match the number of parameters in the macro definition. The liberal use of parentheses ensures that the precedence of complicated arguments is not misinterpreted. If the name of the macro being defined occurs in *replacement-list* (even as a result of another macro expansion), it is not expanded.

The parameters in the *identifier-list* are separated by commas. Each name must be unique. No spaces can separate *identifier* and the opening parenthesis. Use line concatenation (placing a backslash (\) before the newline character) for long directives on multiple source lines. The scope of a parameter name extends to the new line that ends *replacement-list.*

This example illustrates the **#define** directive for manifest constants and macros:

```
#define WIDTH       80              /* Manifest constant */
#define LENGTH      ( WIDTH + 10 ) /* Macro             */
```

The first statement 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 statements 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.

Arguments with side effects sometimes cause macros to produce unexpected results. A given parameter may appear more than once in *replacement-list.* If that parameter is replaced by an expression with side effects, the expression, with its side effects, may be evaluated more than once (see examples in "Token-Pasting Operator" on page 197).

A **#define** without a *replacement-list* removes occurrences of *identifier* from the source file. The *identifier* is still considered defined, however, and yields the value 1 when tested with the **#if defined** directive (discussed in "The defined Operator" on page 204). A second **#define** for the same identifier generates an error unless the second token sequence is identical to the first.

The **#undef** directive causes an identifier's preprocess definition to be removed. See "The #undef Directive" on page 198.

Microsoft Specific

Microsoft C version 6.0 allows a macro to be redefined provided it is lexically identical to the previous definition. ANSI C considers macro redefinition an error. C 7.0 allows this behavior but generates a warning. For example these macros are equivalent for C 7.0 but generate warnings since ANSI C considers this an error.

```
#define test( f1, f2 ) ( f1 * f2 )
#define test( a1, a2 ) ( a1 * a2 )
```

Defining macros and constants with the /D command-line option has the same effect as using a **#define** preprocessing directive at the beginning of your file. Up to 30 macros can be defined with the /D option. See page 206, "The #ifdef and #ifndef Directives," for more information about defining constants from the command line. ♦

Preprocessor Operators

Four preprocessor-specific operators are used in the context of the **#define** directive. This list gives a summary of each. The first three preprocessor operators are discussed in the next three sections. The fourth, the **defined** operator, is discussed on page 204.

Microsoft Specific

Charizing operator (#@)
  Causes the corresponding argument to be enclosed in single quotation marks and to be treated as a character. ♦

Stringizing operator (#)
  Causes the corresponding argument to be enclosed in double quotation marks.

Token-pasting operator (##)
  Allows tokens used as arguments to be concatenated to form other tokens.

defined operator
  Simplifies the writing of compound expressions in certain macro directives. Used as part of a constant expression that can be tested in an **#if** block to determine if a particular identifier has been defined as a macro.

Stringizing Operator (#)

The number-sign or "stringizing" operator (#) converts macro parameters (after expansion) to string constants. It is used only with macros that take arguments. If it precedes a parameter in the macro definition, the argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal. The string literal then replaces each occurrence of a combination of the stringizing operator and parameter within the macro definition.

White space preceding the first token of the argument and following the last token of the argument is ignored. Any white space between the tokens in the argument is reduced to a single white space in the resulting string literal. Thus, if a comment occurs between two tokens in the argument, it is reduced to a single white space. The resulting string literal is automatically concatenated with any adjacent string literals from which it is separated only by white space.

Further, if a character contained in the argument usually requires an escape sequence when used in a string literal—for example, the quotation-mark (“) or backslash (\) characters—the necessary escape backslash is automatically inserted before the character.

The following example shows a macro definition that includes the stringizing operator and a **main** function that invokes the macro:

```
#define stringer( x ) printf( #x "\n" )

main()
{
    stringer( In quotes in the printf function call\n );
    stringer( "In quotes when printed to the screen"\n );
    stringer( "This: \"  prints an escaped double quotation mark" );
}
```

Such invocations would be expanded during preprocessing, producing the following code:

```
main()
{
printf( "In quotes in the printf function call\n" "\n" );
printf( "\"In quotes when printed to the screen\"\n" "\n" );
printf( "\"This: \\\" prints an escaped double quotation mark\"" "\n" );
}
```

When the program is run, screen output for each line would be as follows:

```
In quotes in the printf function call

"In quotes when printed to the screen"

"This: \" prints an escaped double quotation mark"
```

To debug macros, compile your program with the /P command-line option. This preprocesses the source file and sends the output to a file.

Microsoft Specific The Microsoft extension to the ANSI C standard that allowed expanded formal macro arguments to appear inside of string literals and character constants is no longer supported. Code that relied on this extension should be rewritten using the stringizing (#) operator.♦

Charizing Operator (#@)

Microsoft Specific The charizing operator can be used only with the arguments of macros. If a #@ precedes a parameter in the definition of the macro, the argument is enclosed in single quotation marks and treated as a character when the macro is expanded. For example,

```
#define makechar(x)  #@x
```

causes the statement

```
a = makechar(b);
```

to be expanded into

```
a = 'b';
```

The single-quotation character cannot be used with the charizing operator.♦

Token-Pasting Operator (##)

The double-number-sign or "token-pasting" operator (##) (sometimes called the "merging" operator) is used in both object-like and function-like macros. It permits separate tokens to be joined into a single token, and therefore cannot be the first or last token in the macro definition.

If a parameter in a macro definition is preceded or followed by the token-pasting operator, the parameter is immediately replaced by the unexpanded argument. Macro expansion is not performed on the argument prior to replacement.

Then, each occurrence of the token-pasting operator in *replacement-list* is removed, and the tokens preceding and following it are concatenated. The resulting token must be a valid token. If it is, the token is rescanned for possible replacement if it represents a macro name. The identifier name following the **#define** preprocessing directive 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 preceding or following the operator is optional.

This example illustrates use of both the "stringizing" and "token-pasting" operators in specifying program output.

```
#define paster( n ) printf( "token" #n " = %d", token##n )
int token9;
```

The macro is called with a numeric argument such as:

```
paster( 9 );
```

The macro yields

```
printf( "token" "9" " = %d", token9 );
```

which becomes

```
printf( "token9 = %d", token9 );
```

The #undef Directive

The **#undef** directive removes (undefines) a macro name previously created with **#define**.

Syntax

control-line :
  #undef *identifier new-line*

The **#undef** directive removes the current definition of *identifier*. Consequently, subsequent occurrences of *identifier* are ignored by the preprocessor. 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. Macro replacement is not performed within **#undef** directives.

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 "The #if, #elif, #else, and #endif Directives" on page 202) to control conditional compilation of the source program.

```
#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.

Predefined Macros

ANSI C recognizes five predefined macros, and the Microsoft C compiler extensions provide several more. The names of the ANSI predefined macros begin and end with two underscores. These macros take no arguments, and cannot be redefined.

The ANSI-recognized predefined macros are:

Macro	Description
__DATE__	The translation date of the current source file. The date is a character string of the form `Mmm dd yyyy`. The month name `Mmm` is the same as for dates generated by the library function **asctime** declared in TIME.H. When the operating system does not provide the date, the default value for **__DATE__** is `MAY 03 1957`.
__FILE__	The name of the current source file. **__FILE__** expands to a string surrounded by double quotation marks.
__LINE__	The line number in the current source file. The line number is a decimal integer constant. It can be altered with a **#line** directive.
__STDC__	Indicates full conformance with the ANSI C standard. Defined as the integer constant 1 only if the /Za command-line option is given. Not defined under /Ze.
__TIME__	The translation time of the current source file. The time is a character string of the form `hh:mm:ss`. When the operating system does not provide the time, the default value for **__TIME__** is `17:00:00`.

Microsoft Specific

The Microsoft-specific predefined macros are described in the following list. Both forms of a predefined identifier (with and without an underscore) are defined if you specify the /Ze command line option (or, from within PWB, you must select Microsoft Extensions from the Additional Global Options dialog box from the C Compiler Options dialog box).

Identifier	Function
MSDOS, **_MSDOS**	Always defined. Identifies target operating system as MS-DOS.
M_I86, **_M_I86**	Always defined. Identifies target machine as a member of the 8086 family.
M_I8086, **_M_I8086**	Defined for 8086 and 8088 processors (default or /G0 option).
M_I286, **_M_I286**	Defined for 80286 processor (/G2 option).
M_I386, **_M_I386**	Defined for 80386 processor (/G3 option).
M_I86mM, **_M_I86mM**	Always defined. Identifies memory model, where ‘m’ is either T (tiny model), S (small model), C (compact model), M (medium model), L (large model), or H (huge model). If huge model is used, both **M_I86LM** and **M_I86HM** are defined. Small model is the default.
_MSC_VER	Defines the compiler version in the form: `ddd`. Defined as 700 for Microsoft C/C++.
_CHAR_UNSIGNED	Defined only when the /J option is given to make **char** unsigned by default.

Identifier	Function
__TIMESTAMP__	The date and time of the last modification of the source file, expressed as a string literal in the form: `"Ddd Mmm hh:mm:ss yyyy"`.
_PCODE	Translated to pcode. Defined when /Oq is enabled.
_QC	Supported for compatibility with Microsoft C version 6.0. The **_FAST** macro is the recommended alternative.
__cplusplus__	Reserved by the Microsoft C/C++ version 7.0 compiler to assist in portability between ANSI C and C++.
_FAST	Supersedes **_QC**, which is still supported but not recommended.

The /Ze command-line option, the default for Microsoft C, enables the Microsoft extensions. The /Za command-line option, compiling for ANSI compatibility, defines only the identifier form that has a leading underscore.♦

7.4 Include Files

The **#include** directive tells the preprocessor to treat the contents of the named file as if it appeared in the source program at the point where the directive appears. These files are called "header files." You can organize constant and macro definitions into header 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.

Syntax

control-line :
  #include "*path-spec*" *new-line*   /* Programmer-supplied header files */
  #include <*path-spec*> *new-line*   /* Standard C header files */

Both forms cause replacement of the **#include** directive by the contents of the source file given. The first form is usually used for header files that you write. The second form is used for the standard C header files.

The *path-spec* is a filename optionally preceded by a directory specification. The filename must name an existing file. The syntax of the *path-spec* depends on the operating system on which the program is compiled.

Microsoft Specific

The preprocessor stops searching as soon as it finds a file with the given name. If you specify a complete, unambiguous path specification for the include file, between two sets of double quotation marks (" "), the preprocessor searches only that path specification and ignores the standard directories. Unambiguous path specifications are called "fully qualified." A fully specified filename is a filename where the first character is a forward slash (/) or a backslash (\) or the second character is a colon (**:**). For example, F:\C7\SPECIAL\INCL\TEST.H) is a fully qualified path specification.

If the filename enclosed in double quotation marks is an incomplete (or "relative") path specification, the preprocessor first searches the "parent" file's directory. A parent file is the file containing the **#include** directive. For example, if you include a file named `file2` within a file named `file1`, `file1` is the parent file.

Include files can be "nested"; that is, an **#include** directive can appear in a file named by another **#include** directive. For example, `file2` could include `file3`. In this case, `file1` would still be the parent of `file2` but would also be the "grandparent" of `file3`.

For include files specified as **#include** "*path-spec*", directory searching begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename were enclosed in angle brackets.

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. Instead, it begins by searching for the file in the directories specified on the compiler command line following /I. If the /I option is not present or fails, the preprocessor uses the INCLUDE environment variable to find any include files within angle brackets. The INCLUDE environment variable can contain multiple paths separated by semicolons (**;**). If more than one directory appears as part of the /I option or within the INCLUDE environment variable, the preprocessor searches them in the order they appear.

Nesting of include files is limited by the available memory. Once the nested **#include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

For example, the command

```
CL /ID:\MSC\INCLUDE MYPROG.C
```

causes the preprocessor to search the directory D:\MSC\INCLUDE for include files such as STDIO.H. The commands

```
SET INCLUDE = D:\MSC\INCLUDE
CL MYPROG.C
```

have the same effect. If both sets of searches fail, a fatal error is generated.

Note Programs containing references to fully specified include files may not compile on other computers. ♦

7.5 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 are passed on to the compiler and which text blocks are removed from the source file during preprocessing.

The #if, #elif, #else, and #endif Directives

The **#if** directive, together with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file. If the expression you write (after the **#if**) has a nonzero value, the statements immediately following the **#if** directive are retained in the translation unit.

Syntax

preprocessing-file :
  group opt

group :
  group-part
  group group-part

group-part :
  pp-tokens opt *new-line*
  if-section
  control-line

if-section :
  if-group elif-groups opt *else-group* opt *endif-line*

if-group :
  #if *restricted-constant-expression new-line group* opt
  #ifdef *identifier new-line group* opt
  #ifndef *identifier new-line group* opt

elif-groups :
  elif-group
  elif-groups elif-group

elif-group :
  #elif *restricted-constant-expression new-line group* opt

else-group :
  #else *new-line group* opt

endif-line :
  #endif *new-line*

control-line :
  #include "*path-spec*" *new-line* /* Programmer-supplied header files */
  #include <*path-spec*> *new-line* /* Standard C header files */
  #define *identifier replacement-list new-line* /* Macro with arguments */
  #define *identifier* (*identifier-list*$_{opt}$ new-line) *replacement-list*
  #undef *identifier new-line*
  #line *digit-sequence new-line*
  #line *digit-sequence* "*filename*" $_{opt}$ *new-line*
  #line *digit-sequence preprocessing-tokens new-line*
  #error *preprocessor-tokens* $_{opt}$ *new-line*
  #pragma *pragma-directive* $_{opt}$ *new-line*
  # *new-line*

replacement-list :
  pp-tokens $_{opt}$

pp-tokens :
  preprocessing-token
  pp-tokens preprocessing-token

preprocessing-token :
  header-name
  identifier
  pp-number
  character-constant
  string-literal
  operator
  punctuator
  each nonwhite-space character that cannot be one of the above

new-line :
  the newline character

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 **#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 unmatched **#if** directive.

The preprocessor selects a single *group* by evaluating the restricted constant expression following each **#if** or **#elif** directive until it finds a true (nonzero) restricted constant expression. It selects all text (including other preprocessor directives beginning with #) up to its associated **#elif**, **#else**, or **#endif**.

The constant expressions used with the **#if** directives are called *restricted-constant-expressions* since some restrictions apply here that do not apply to all constant expressions. The preprocessor processes the selected *group* and passes it to the compiler. If *group* contains preprocessor directives, the preprocessor carries out those directives. Any text blocks not selected by the preprocessor are not compiled.

If all occurrences of *restricted-constant-expression* are false, or if no **#elif** directives appear, the preprocessor selects the text block after the **#else** clause. If the **#else** clause is omitted, and all instances of *restricted-constant-expression* in the **#if** block are false, no text block is selected.

The Restricted Constant Expression

The *restricted-constant-expression* is an integer constant expression with these additional restrictions:

- All expressions must have integral type and can only include integer constants and character constants.
- The expression cannot use **sizeof** or a type cast operator, nor can assignment operators and the sequential evaluation operator (,) be used in the constant expression.
- The translation represents type **int** the same as type **long** and **unsigned int** the same as **unsigned long**.
- The compiler can translate character constants to a set of code values different from the set for the target environment. To determine the properties of the target environment, check values of macros from LIMITS.H in an application built for the target environment. The target environment may not be able to represent all ranges of integers.
- The expression must not perform any environmental inquiries and must remain insulated from implementation details on the target computer.

The defined Operator

The *restricted-constant-expression* can contain the preprocessor operator **defined** as shown:

defined(*identifier*)
defined *identifier*   /* Alternative equivalent form */

This constant expression is considered true (nonzero) if the *identifier* is currently defined as a macro; otherwise, the condition is false (0). An identifier defined as empty text is considered defined. The **defined** directive can be used in a **#if** and a **#elif**, but nowhere else.

In the following example, the **#if** and **#endif** directives control compilation of one of three function calls.

```
#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

The function call to `credit` is compiled if the macro `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.

The following conditional compilation statements assume a constant named `DLEVEL` has already been defined. If an identifier used with **#if** has not been defined, the identifier evalutes to 0.

```
#if DLEVEL > 5                /* First example */
    #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


#if DLEVEL == 0                /* Second example */
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif
```

The **#if** block in the first example 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 statements after the **#else** are processed.

The **#elif** and **#else** directives in the second example are used to make one of four choices, based on the value of `DLEVEL`. The constant `STACK` is set to 0, 100, or 200, depending on the definition of `DLEVEL`. If `DLEVEL` is greater than 5, then the statement

```
#elif DLEVEL > 5
display(debugptr);
```

is compiled and `STACK` is not defined.

Microsoft Specific

The *identifier* can be passed from the command line using the /D option. Up to 30 macros can be specified with /D.

This is useful for checking if a definition exists since a definition can be passed from the command line. For example,

```
#if !defined test    /* These three statements go in your source code */
#define final
#endif

CL /Dtest       /* This is the command for compilation */
```

In this example, a macro named `final` is defined if `test` has not been defined. You can enter either `test` or `final` from the command line at compilation time. The example above shows `test` being entered from the command line. Alternatively, you can use PWB. From the PWB Options menu, choose Language Options. Then select Additional Global Options or Additional Debug Options and type the constants for your program.

Conditional compilation expressions are treated as **signed long** values. For example, this expression is true:

```
#if 0xFFFFFFFFL > 1UL
```

These expressions are evaluated using the same rules as expressions in C. ♦

The #ifdef and #ifndef Directives

The **#ifdef** and **#ifndef** directives perform the same task as the **#if** directive when it is used with **defined**(*identifier*).

Syntax

if-group :
#ifdef *identifier new-line group* opt
#ifndef *identifier new-line group* opt

the syntax above is equivalent to

#if defined *identifier*
#if !defined *identifier*

You can use the **#ifdef** and **#ifndef** directives anywhere **#if** can be used. The **#ifdef** *identifier* statement is equivalent to `#if 1` when *identifier* has been defined and is equivalent to `#if 0` when *identifier* has not been defined, has been undefined with the **#undef** directive, or has been defined with the value zero. These directives check only for identifiers defined with **#define** (or **#undefined**), not for identifiers declared in the C source code.

The **#ifdef** and **#ifndef** directives are provided mainly for compatibility with previous versions of the compiler. The preferred form is an **#if** directive and a **defined** *identifier* constant expression since more complex expressions can be used with **#if defined**.

7.6 Line Control

The **#line** directive changes the compiler's internally stored line number and filename to a given line number and filename. The compiler uses the line number and filename to refer to errors that it finds during compilation. The line number usually refers to the current input line, and the filename refers to the current input file. The line number is incremented after each line is processed. The syntax is

Syntax

control-line :
  #line *digit-sequence new-line*
  #line *digit-sequence* **"***filename***"** opt *new-line*
  #line *digit-sequence preprocessing-tokens new-line*

The *digit-sequence* value in the **#line** directive can be any integer constant in the range 1 through 32,767. The *filename* can be any combination of characters and must be enclosed in double quotation marks (“ ”). If *filename* is omitted, the previous filename remains unchanged. In the third line of syntax above, macro replacement is performed on the preprocessing tokens and the result must match one of the two previous lines.

You can alter the source line number and filename by writing a **#line** directive. The translator uses the line number and filename to determine the values of the predefined macros **__FILE__** and **__LINE__**.

The current line number and filename are always available through the predefined macros **__LINE__** and **__FILE__**. You can use the **__LINE__** and **__FILE__** identifiers to insert self-descriptive error messages into the program text.

The **__FILE__** macro expands to a string whose contents are the filename, surrounded by double quotation marks (" "). See "Predefined Macros" on page 198.

If you change the line number and filename, the compiler ignores the previous values and continues processing with the new values. The **#line** directive is typically used by program generators to cause error messages to refer to the original source file instead of to the generated program.

These examples illustrate **#line** and the **__LINE__** and **__FILE__** macros.

```
#line 151 "copy.c"
```

In this statement, the internally stored line number is set to 151 and the filename is changed to `copy.c`.

```
#define ASSERT(cond)

if( !(cond) )\
{printf( "assertion error line %d, file(%s)\n", \
__LINE__, __FILE__ );}
```

In this example, 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.

7.7 Error Directives

Error directives produce compile-time error messages.

Syntax

control-line :
  #error *preprocessor-tokens* $_{\text{opt}}$ *new-line*

The error messages include the argument *preprocessor-tokens*, which is subject to macro expansion. These directives are most useful for detecting programmer inconsistencies and violation of constraints during preprocessing. The following example demonstrates error processing during preprocessing.

```
#if !defined(error_chk)
#error No error checking enabled.
#endif
```

If the `error_chk` constant is not defined at compile time, the

```
No error checking enabled
```

message prints on the screen along with other compiler messages.

7.8 Pragma Directives (Microsoft Specific)

Although portability is a hallmark of C, its creators recognized that every C compiler needs to support some features unique to its host machine. Some programs, for instance, need to exercise precise control over the memory areas where data is placed or to control the way certain functions receive parameters. The **#pragma** directive offers a way for each C compiler to offer machine-specific features while retaining overall compatibility with the C language and other C compilers.

Since pragmas are machine-specific by definition, they are usually different for every C compiler. Pragmas can be used in conditional statements to provide specific preprocessor functionality or to provide implementation-defined information to the compiler. The pragmas discussed in this section apply to the Microsoft C compiler.

Syntax

#pragma *pragma-directive* $_{\text{opt}}$ *new-line*

The *pragma-directive* is one of a series of directives that gives a specific compiler instruction and arguments, if any. The number sign (**#**) must be the first nonwhite-space character on the line containing the pragma; white-space characters can separate the number sign and the word **pragma**. The argument to **#pragma** is subject to macro expansion.

With many of these pragmas, one of the arguments can be left out. When that is done, the setting of the option returns to the command-line setting. For example:

```
#pragma pack(1)

struct var
{
    /* Structure elements defined here */
}

#pragma pack( )
```

In this example, the **pack** pragma forces the structure to be packed on one-byte boundaries until `#pragma pack( )` tells the compiler to go back to whatever the previous setting had been.

The **#pragma** directives instruct the compiler to implement the features specified by the argument. The Microsoft C compiler recognizes the following pragmas:

alloc_text	**data_seg**	**linesize**	**pagesize**
auto_inline	**function**	**message**	**skip**
check_pointer	**hdrstop**	**native_caller**	**subtitle**
check_stack	**check_stack**	**optimize**	**title**
code_seg	**inline_recursion**	**pack**	**warning**
comment	**intrinsic**	**page**	

These pragma directives are summarized in the following list.

#pragma alloc_text(*textsegment,function1,* **...)**
Names the segment where the specified routine definitions are to reside. This must occur between a function declarator and the function definition for the named functions. The **alloc_text** pragma is still supported, but the recommended technique is to use **__based**.

#pragma auto_inline ([[on | off]])
Inhibits the inline expansion of a function. The **auto_inline** pragma inhibits the preprocessor from expanding a function when the /Ob2 command-line option is in effect. To use it, place one pragma just before and again just after a function definition.

#pragma check_pointer ([[{ on | off }]])
Instructs the compiler to turn off pointer checking if **off** is specified, or to turn on pointer checking if **on** is specified. Checks every pointer dereference to make sure the pointer is not a null or out-of-range pointer. Also enabled with the /Zr command-line option, which is only available with the /qc option. (Within PWB select Additional Debug Options from the C Compiler Options dialog box. Then select Quick Compile and Null Pointer Checking.)

#pragma check_stack ([[{ on | off }]])
Instructs the compiler to turn off stack probes if **off** is specified, or to turn on stack probes if **on** is specified. If no argument is given, stack probes are treated according to the default (on, unless /Gs was used). You can reduce the size of a program and speed up execution slightly by removing stack probes. You can do this with either the /Gs option or the **check_stack** pragma.

#pragma code_seg ([[" *segment_name*" **[[, "** *segment_class* **"]])**
Specifies a segment where functions are to be allocated, allowing the use of based allocation without rewriting code. Using **#pragma code_seg** is equivalent to using **__based** when **__based** is used for allocation. You can specify the class for the segment by giving the *segment_class* as a string. Using **#pragma code_seg** without a *segment_name* string resets allocation to whatever it was when compilation began.

#pragma comment (*comment-type* **[[,** *commentstring***]])**
Allows you to place a comment record in an object file or executable file. The *comment-type* specifies the type of comment record. The optional *commentstring* is a string literal that provides additional information for some comment types. Because *comment-type* is a string literal, it obeys all the rules

for string literals with respect to escape characters, embedded quotation marks ("), and concatenation.

#pragma data_seg ([[" *segment_name*" [[**,** " *segment_class* "]])
Specifies a segment where data is to be allocated, allowing the use of based allocation without rewriting code. Using **#pragma data_seg** is equivalent to using **__based** when **__based** is used for allocation. The *segment-class* allows you to assign a segment class to a segment name. This pragma applies only to initialized data and does not affect tentative definitions. Using **#pragma data_seg** without a *segment_name* string resets allocation to whatever it was when compilation began. This **data_seg** pragmas is not equivalent to the **data_seg** pragma supported by earlier versions of the compiler.

#pragma function (*function1* [[,*function2*, ...]])
Specifies that calls to the specified functions will actually take place. The **intrinsic** pragma affects a specified function beginning where the pragma appears. The effect continues to the end of the source file or to the appearance of a **function** pragma specifying that function.

#pragma hdrstop [[("*filename*")]]
Controls the way precompiled headers work. The *filename* is the name of the precompiled header file to use or create (depending on compilation options). If *filename* does not contain a path specification, the precompiled header file is assumed to be in the same directory as the source file. See Chapter 2 in the *Programming Techniques* manual for more information on precompiled header files.

#pragma inline_depth ([[0..255]])
Controls the number of times that inline expansion can occur by controlling the number of times that a series of function calls can be expanded (from 0 to 255 times). Use this pragma to control inline functions, or functions that the compiler automatically expands under the /Ob2 option. Requires an /Ob command-line option setting of either 1 or 2.

#pragma inline_recursion ([[**on** | **off**]])
Controls the inline expansion of direct or mutually recursive function calls. Use this pragma to control inline functions, or functions that the compiler automatically expands under the /Ob2 option. Requires an /Ob command-line option setting of either 1 or 2. The default state for **inline_recursion** is **off**.

#pragma intrinsic (*function1* [[,*function2*, ...]])
Specifies that calls to the specified functions are intrinsic (a library function known to the compiler). Alternatively, you can use the /Oi option to make intrinsic the default form for functions that have intrinsic forms. In this case, you can use the **function** pragma to override /Oi for specified functions. This pragma cannot be used with /qc.

The following functions have intrinsic forms:

_alloca	**_rotl**	**log10**	**_asinl**
_disable	**_rotr**	**memcmp**	**_atanl**
_enable	**_strset**	**memcpy**	**_atan2l**
_fmemcmp	**abs**	**memset**	**_ceill**
_fmemcpy	**acos**	**pow**	**_cosl**
_fmemset	**asin**	**sin**	**_coshl**
_fstrcat	**atan**	**sinh**	**_expl**
_fstrcmp	**atan2**	**sqrt**	**_floorl**
_fstrcpy	**ceil**	**strcat**	**_fmodl**
_fstrlen	**cos**	**strcmp**	**_logl**
_fstrset	**cosh**	**strcpy**	**_log10l**
_inp	**exp**	**strlen**	**_powl**
_inpw	**fabs**	**tan**	**_sinl**
_lrotl	**floor**	**tanh**	**_sinhl**
_lrotr	**fmod**	16-Bit Target	**_sqrtl**
_outp	**labs**	Only:	**_tanl**
_outpw	**log**	**_acosl**	**_tanhl**

#pragma linesize ([[*num-chars*]] **)**
Specifies the number of characters per line in the source listing. The optional parameter *num-chars* is an integer constant in the range 79–132. If *num-chars* is absent, the compiler uses the value specified in the /Sl option or, if that option is absent, the default value of 79 characters per line. The **linesize** pragma takes effect the line after it appears.

#pragma message (*messagestring* **)**
Sends a string literal to the standard output without terminating the compilation. The *messagestring* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

#pragma native_caller ([[{ **on** | **off** }]]**)**
Controls the removal of native-code entry points from within source code. If you have p-code functions that are called only by other p-code functions, you can omit those entry points and save those bytes by using the /Gn compiler option or on a function-by-function basis with this pragma. See Chapter 3 in the *Programming Techniques* manual for more information on p-code.

#pragma optimize ("[[*optimization-option-list*]]", { **off** | **on** } **)**
Specifies optimizations to be performed. This pragma must appear outside of a function. The optimization option list may be zero or more of the following: a, c, e, g, l, n, p, q, t, and w. These letters correspond to the /O compiler options.

#pragma pack ([[{ **1** | **2** | **4** }]] **)**
Specifies the byte boundary for packing members of C structures. You can use the /Zp option to specify the same packing for all structures in a module. The default is 2 for 16-bit computers and 4 on 32-bit computers. See "Structure Declarations," on page 65 for more information.

32-Bit Specific

On 32-bit targets, the packing can be set at 8 or 16 as well as 1, 2, and 4 as given for 16-bit targets.♦

#pragma page ([[*numpages*]])
Tells the compiler to generate formfeeds in the source listing at the line where it appears. The **page** pragma generates one or more formfeeds (page eject) in the source listing (created with /Fs) at the place where the pragma appears. The number of formfeeds is specified by *numpages*. Legal values are 1–127, with the default being 1.

#pragma pagesize ([[*numlines*]])
Sets the number of lines per page in the source listing. The optional *numlines* parameter is an integer constant in the range 15–255 that specifies the number of lines you want on each page of the source listing to have. If *numlines* is absent, the pragma sets the page size to the number of lines specified in the /Sp option or, if that option is absent, to a default value of 63 lines.

#pragma skip ([[*numlines*]])
Skips the specified number of lines in the source listing. The **skip** pragma generates one or more newline characters (carriage return–linefeed) in the source listing at the point where the pragma appears. The optional *numlines* parameter is an integer constant in the range 1–127 that specifies the number of lines to skip. If this parameter is absent, the **skip** pragma defaults to one line.

#pragma subtitle ("*subtitlename*")
Specifies a subtitle for the source listing. The *subtitlename* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination. A null *subtitlename* erases any previous subtitle.

#pragma title ("*titlename*")
Specifies a title for the source listing. The title appears in the upper-left corner of each page of the listing. The *titlename* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination. A null *titlename* erases any previous title.

#pragma warning(*warning-specifier*[[; *warning-specifier*]])
Controls the warning level for compiler errors. The *warning-specifier* has the syntax *warning-type*:*warning-number-list* where *warning-type* can be **once**, **default**, **1**, **2**, **3**, **4**, **disable**, or **error**. The specifier **once** tells the compiler to display a warning only once, **default** is the standard compiler warning level, and **1–4** force warning levels 1–4. Specifying **disable** disables the selected warnings, and **error** forces a warning to be reported as an error. The *warning-number-list* can be any number between 1 and 699 or 4001 and 4699.

The **loop_opt** pragma has been replaced with the **optimize** pragma. The **same_seg** pragma is no longer supported. Use **__based** for specifying the placement of external variables in memory.

Appendixes

C Language Syntax Summary

Appendix A

This appendix gives the full description of the C language and the Microsoft-specific C language features. You can use the syntax notation in this appendix to determine the exact syntax for any language component. The explanation for the syntax appears in the section of this manual where a topic is discussed.

Note This syntax summary is not part of the ANSI standard, but is included for information only. Microsoft-specific syntax is noted in comments following the syntax.

Definitions

Terminals are endpoints in a syntax definition. No other resolution is possible. Terminals include the set of reserved words and user-defined identifiers.

Nonterminals are placeholders in the syntax and are defined elsewhere in this syntax summary. Definitions can be recursive.

An optional component is indicated by the subscripted $_{\text{opt}}$. For example

{ *expression* $_{\text{opt}}$ }

indicates an optional expression enclosed in curly braces.

Conventions

The conventions use different font attributes for different components of the syntax. The symbols and fonts are as follows:

Attribute	Description
nonterminal	Italic type indicates nonterminals.
const	Terminals in boldface type are literal reserved words and symbols that must be entered as shown. Characters in this context are always case sensitive.
opt	Nonterminals followed by $_{\text{opt}}$ are always optional.
default typeface	Characters in the set described or listed in this typeface can be used as terminals in C statements

A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced with the words "one of."

Syntax Categories

The syntax categories are:

- Lexical Grammar
  - Tokens
  - Keywords
  - Identifiers
  - Constants
  - String Literals
  - Operators
  - Punctuators
- Phrase Structure Grammar
  - Expressions
  - Declarations
  - Statements
  - External Definitions
- Preprocessing Directives

A.1 Language Syntax Summary

Lexical Grammar

Tokens

token :
  keyword
  identifier
  constant
  string-literal
  operator
  punctuator

preprocessing-token :
  header-name
  identifier
  pp-number
  character-constant
  string-literal
  operator
  punctuator
  each nonwhite-space character that cannot be one of the above

Keywords

keyword : one of

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**

Identifiers

identifier :
  nondigit
  identifier nondigit
  identifier digit

nondigit : one of

```
_ 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
```

digit : one of

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

Constants

constant :
  floating-point-constant
  integer-constant
  enumeration-constant
  character-constant

floating-point-constant :
  fractional-constant exponent-part $_{\text{opt}}$ *floating-suffix* $_{\text{opt}}$
  digit-sequence exponent-part floating-suffix $_{\text{opt}}$

fractional-constant :
  digit-sequence $_{\text{opt}}$ **.** *digit-sequence*
  digit-sequence **.**

exponent-part :
  `e` *sign* $_{\text{opt}}$ *digit-sequence*
  `E` *sign* $_{\text{opt}}$ *digit-sequence*

sign : one of
  `+   -`

digit-sequence :
  digit
  digit-sequence digit

floating-suffix : one of
  `f   l   F   L`

integer-constant :
decimal-constant integer-suffix $_{\text{opt}}$
octal-constant integer-suffix $_{\text{opt}}$
hexadecimal-constant integer-suffix $_{\text{opt}}$

decimal-constant :
nonzero-digit
decimal-constant digit

octal-constant :
0
octal-constant octal-digit

hexadecimal-constant :
0x *hexadecimal-digit*
0X *hexadecimal-digit*
hexadecimal-constant hexadecimal-digit

nonzero-digit : one of
1 2 3 4 5 6 7 8 9

octal-digit : one of
0 1 2 3 4 5 6 7

hexadecimal-digit : one of
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

unsigned-suffix : one of
u U

long-suffix : one of
l L

character-constant :
'c-char-sequence'
L*'c-char-sequence'*

integer-suffix :
unsigned-suffix long-suffix $_{\text{opt}}$
long-suffix unsigned-suffix $_{\text{opt}}$

c-char-sequence :
c-char
c-char-sequence c-char

c-char :
  Any member of the source character set except the single quotation mark ('), backslash (\), or newline character
  escape-sequence

escape-sequence :
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence : one of
  \a \b \f \n \r \t \v
  \' \" \\ \?

octal-escape-sequence :
  \ *octal-digit*
  \ *octal-digit octal-digit*
  \ *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence :
  \x *hexadecimal-digit*
  hexadecimal-escape-sequence hexadecimal-digit

String Literals

string-literal :
  "s-char-sequence $_{\text{opt}}$"
  L"*s-char-sequence* $_{\text{opt}}$"

s-char-sequence :
  s-char
  s-char-sequence s-char

s-char :
  any member of the source character set except the double-quote quotation mark ("), backslash (\), or newline character
  escape-sequence

Operators

assignment-operator : one of
  = *= /= %= += –= <<= >>= &= ^= |=

Punctuators

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

Phrase Structure Grammar

Expressions

primary-expression :
identifier
constant
string-literal
(*expression*)

expression :
assignment-expression
expression , *assignment-expression*

constant-expression :
conditional-expression

conditional-expression :
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

assignment-expression :
conditional-expression
unary-expression assignment-operator assignment-expression

postfix-expression :
primary-expression
postfix-expression **[** *expression* **]**
postfix-expression **(** *argument-expression-list* $_{\text{opt}}$ **)**
postfix-expression **.** *identifier*
postfix-expression **–>** *identifier*
postfix-expression **++**
postfix-expression **––**
postfix-expression **:>** *expression* /* Microsoft-specific */

argument-expression-list :
assignment-expression
argument-expression-list , *assignment-expression*

unary-expression :
  postfix-expression
  ++ _unary-expression_
  — _unary-expression_
  unary-operator cast-expression
  sizeof _unary-expression_
  sizeof (_type-name_)

unary-operator : one of
  & * + – ~ !

cast-expression :
  unary-expression
  (_type-name_) _cast-expression_

multiplicative-expression :
  cast-expression
  multiplicative-expression * _cast-expression_
  multiplicative-expression / _cast-expression_
  multiplicative-expression % _cast-expression_

additive-expression :
  multiplicative-expression
  additive-expression + _multiplicative-expression_
  additive-expression – _multiplicative-expression_

shift-expression :
  additive-expression
  shift-expression << _additive-expression_
  shift-expression >> _additive-expression_

relational-expression :
  shift-expression
  relational-expression < _shift-expression_
  relational-expression > _shift-expression_
  relational-expression <= _shift-expression_
  relational-expression >= _shift-expression_

equality-expression :
  relational-expression
  equality-expression == _relational-expression_
  equality-expression != _relational-expression_

AND-expression :
  equality-expression
  AND-expression **&** _equality-expression_

exclusive-OR-expression :
  AND-expression
  exclusive-OR-expression ^ *AND-expression*

inclusive-OR-expression :
  exclusive-OR-expression
  inclusive-OR-expression | *exclusive-OR-expression*

logical-AND-expression :
  inclusive-OR-expression
  logical-AND-expression **&&** *inclusive-OR-expression*

logical-OR-expression :
  logical-AND-expression
  logical-OR-expression || *logical-AND-expression*

Declarations

declaration :
  declaration-specifiers init-declarator-list $_{\text{opt}}$;

declaration-specifiers :
  storage-class-specifier declaration-specifiers $_{\text{opt}}$
  type-specifier declaration-specifiers $_{\text{opt}}$
  type-qualifier declaration-specifiers $_{\text{opt}}$
  attributes $_{\text{opt}}$ *declaration-specifiers* $_{\text{opt}}$ /* Microsoft Specific */

init-declarator-list :
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator :
  declarator
  declarator = *initializer* /* For scalar initialization */

storage-class-specifier :
  auto
  register
  static
  extern
  typedef

type-specifier :
void
char
short
int
long
float
double
signed
unsigned
struct-or-union-specifier
enum-specifier
typedef-name

type-qualifier :
const
volatile

declarator :
pointer opt *direct-declarator*

direct-declarator :
identifier
(*declarator*)
direct-declarator [*constant-expression* opt]
direct-declarator (*parameter-type-list*) /* New-style declarator */
direct-declarator (*identifier-list* opt) /* Obsolete-style declarator */

pointer :
* *type-qualifier-list* opt
* *type-qualifier-list* opt *pointer*

parameter-type-list : /* The parameter list */
parameter-list
parameter-list , **...**

parameter-list :
parameter-declaration
parameter-list , *parameter-declaration*

type-qualifier-list :
type-qualifier
type-qualifier-list type-qualifier

enum-specifier :
enum *identifier* opt { *enumerator-list* }
enum *identifier*

enumerator-list :
  enumerator
  enumerator-list , *enumerator*

enumerator :
  enumeration-constant
  enumeration-constant = *constant-expression*

enumeration-constant :
  identifier

struct-or-union-specifier :
  struct-or-union identifier $_{\text{opt}}$ { *struct-declaration-list* }
  struct-or-union identifier

struct-or-union :
  struct
  union

struct-declaration-list :
  struct-declaration
  struct-declaration-list struct-declaration

struct-declaration :
  specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list :
  type-specifier specifier-qualifier-list $_{\text{opt}}$
  type-qualifier specifier-qualifier-list $_{\text{opt}}$

struct-declarator-list :
  struct-declarator
  struct-declarator-list , *struct-declarator*

struct-declarator :
  declarator
  type-specifier declarator $_{\text{opt}}$: *constant-expression*

parameter-declaration :
  declaration-specifiers declarator /* Named declarator */
  declaration-specifiers abstract-declarator $_{\text{opt}}$ /* Anonymous declarator */

identifier-list : /* For old-style declarator */
  identifier
  identifier-list , *identifier*

abstract-declarator : /* Used with anonymous declarators */
  pointer
  pointer $_{\text{opt}}$ *direct-abstract-declarator*

direct-abstract-declarator :
  (*abstract-declarator*)
  direct-abstract-declarator $_{\text{opt}}$ [*constant-expression* $_{\text{opt}}$]
  direct-abstract-declarator $_{\text{opt}}$ (*parameter-type-list* $_{\text{opt}}$)

initializer :
  assignment-expression
  { *initializer-list* } /* For aggregate initialization */
  { *initializer-list* , }

initializer-list :
  initializer
  initializer-list , *initializer*

type-name :
  specifier-qualifier-list abstract-declarator $_{\text{opt}}$

typedef-name :
  identifier

Statements

Statement :
  labeled-statement
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement

jump-statement :
  goto *identifier* **;**
  continue;
  break;
  return *expression* $_{\text{opt}}$ **;**

compound-statement :
  { *declaration-list* $_{\text{opt}}$ *statement-list* $_{\text{opt}}$ }

declaration-list :
  declaration
  declaration-list declaration

statement-list :
  statement
  statement-list statement

expression-statement :
  expression $_{\text{opt}}$ **;**

iteration-statement :
  while (*expression*) *statement*
  do *statement* **while** (*expression*)
  for (*init-expression* $_{\text{opt}}$ **;** *cond-expression* $_{\text{opt}}$ **;** *loop-expression* $_{\text{opt}}$) *statement*

selection-statement :
  if (*expression*) *statement*
  if (*expression*) *statement* **else** *statement2*
  switch (*expression*) *statement*

labeled-statement :
  identifier : *statement*
  case *constant-expression* **:** *statement*
  default : *statement*

External Definitions

translation-unit :
  external-declaration
  translation-unit external-declaration

external-declaration : /* Allowed only at external (file) scope */
  function-definition
  declaration

function-definition : /* Declarator here is the function declarator */
  declaration-specifiers $_{\text{opt}}$ *declarator declaration-list* $_{\text{opt}}$ *compound-statement*

Preprocessing Directives

preprocessing-file :
 group $_{\text{opt}}$

group :
 group-part
 group group-part

group-part :
 pp-tokens $_{\text{opt}}$ *new-line*
 if-section
 control-line

if-section :
 if-group elif-groups $_{\text{opt}}$ *else-group* $_{\text{opt}}$ *endif-line*

if-group :
 #if *restricted-constant-expression new-line group* $_{\text{opt}}$
 #ifdef *identifier new-line group* $_{\text{opt}}$
 #ifndef *identifier new-line group* $_{\text{opt}}$

elif-groups :
 elif-group
 elif-groups elif-group

elif-group :
 #elif *restricted-constant-expression new-line group* $_{\text{opt}}$

else-group :
 #else *new-line group* $_{\text{opt}}$

endif-line :
 #endif *new-line*

control-line :
 #include "*path-spec*" *new-line* /* Programmer-supplied header files */
 #include <*path-spec*> *new-line* /* Standard C header files */
 #define *identifier replacement-list new-line* /* Macro without parameters */
 #define *identifier* (*identifier-list*$_{\text{opt}}$) *replacement-list new-line*
 #undef *identifier new-line*
 #line *digit-sequence new-line*
 #line *digit-sequence* "*filename* " $_{\text{opt}}$ *new-line*
 #line *digit-sequence preprocessing-tokens new-line*
 #error *preprocessor-tokens* $_{\text{opt}}$ *new-line*
 #pragma *pragma-directive* $_{\text{opt}}$ *new-line*
 # *new-line*

replacement-list:
  pp-tokens $_{\text{opt}}$

new-line :
  the newline character

pp-tokens :
  preprocessing-token
  pp-tokens preprocessing-token

Implementation-Defined Behavior

Appendix B

The American National Standards Institute (ANSI) Standard for the C programming language, volume x3.159-1989, contains an appendix called "Portability Issues." The ANSI appendix lists areas of the C language that ANSI leaves open to each particular implementation. This appendix describes how Microsoft C handles these implementation-defined areas of the C language.

This appendix follows the same order as the ANSI Standard appendix. Each item covered includes references to the ANSI chapter and section that explains the implementation-defined behavior.

Note This appendix describes the U.S. English-language version of the C compiler only. Implementations of Microsoft C for other languages may differ slightly.

B.1 Translation

Diagnostics

How a diagnostic is identified (§2.1.1.3)

Microsoft C produces error messages in the form:

filename(*line-number*) **:** *diagnostic* **C***number message*

where *filename* is the name of the source file in which the error was encountered; *line-number* is the line number at which the compiler detected the error; *diagnostic* is either "error" or "warning"; *number* is a unique four-digit number (preceded by a **C**) that identifies the error or warning; *message* is an explanatory message.

B.2 Environment

Arguments to main

The semantics of the arguments to main (§2.1.2.2.1)

In Microsoft C, the function called at program startup is called **main**. There is no prototype declared for **main**, and it can be defined with zero, two, or three parameters:

```
int main( void )
int main( int argc, char *argv[] )
int main( int argc, char *argv[], char *envp[] )
```

The third line above, where **main** accepts three parameters, is a Microsoft extension to the ANSI standard. The third parameter, **envp**, is an array of pointers to environment variables. The **envp** array is terminated by a null pointer. See "The main Function and Program Execution" on page 30 for more information about **main** and **envp**.

The variable **argc** never holds a negative value.

The array of strings ends with **argv[argc]**, which contains a null pointer.

All elements of the **argv** array are pointers to strings.

A program invoked with no command-line arguments will receive a value of one for **argc**, as the name of the executable file is placed in **argv[0]**. (In DOS versions prior to 3.0, the executable-file name is not available. The letter "C" is placed in **argv[0]**.) Strings pointed to by **argv[1]** through **argv[argc – 1]** represent program parameters.

The parameters **argc** and **argv** are modifiable and retain their last-stored values between program startup and program termination.

Interactive Devices

What constitutes an interactive device (§2.1.2.3)

Microsoft C defines the keyboard and the display as interactive devices.

B.3 Identifiers

Significant Characters Without External Linkage

The number of significant characters without external linkage (§3.1.2)

Identifiers are significant to 247 characters. The compiler does not restrict the number of characters you can use in an identifier; it simply ignores any characters beyond the limit.

Significant Characters with External Linkage

The number of significant characters with external linkage (§3.1.2)

Identifiers declared **extern** in programs compiled with Microsoft C are significant to 247 characters. You can modify this default to a smaller number using the /H (restrict length of external names) option.

Uppercase and Lowercase

Whether case distinctions are significant (§3.1.2)

Microsoft C treats identifiers within a compilation unit as case sensitive. Externally linked identifiers may or may not be case sensitive, depending on whether you use /NOIGNORECASE option when you invoke the linker. The default for the linker is to ignore case, making externally linked identifiers case insensitive.

Thus, symbols in source files are sensitive to case. By default, symbols in object files are not.

Two CL command-line options affect case sensitivity:

- The /Gc (generate Pascal-style function calls) command-line option converts all external identifiers (including function names) to uppercase. The **__pascal** declarator performs the same operation on a function-by-function basis for 16-bit targets.
- The /Zc (compile case insensitive) ignores case at the source level for any identifier names declared with the **__pascal** keyword.

B.4 Characters

The ASCII Character Set

Members of source and execution character sets (§2.2.1)

The source character set is the set of legal characters that can appear in source files. For Microsoft C, the source character set is the standard ASCII character set.

Warning Because keyboard and console drivers can remap the character set, programs intended for international distribution should check the country code.

Multibyte Characters

Shift states for multibyte characters (§2.2.1.2)

Multibyte characters are used by some implementations, including Microsoft C version 7.0, to represent foreign-language characters not represented in the base character set. However, Microsoft C version 7.0 does not support any state-dependent encodings. Therefore, there are no shift states. See "Multibyte and Wide Characters" on page 8 for more information.

Bits per Character

Number of bits in a character (§2.2.4.2.1)

The number of bits in a character is represented by the manifest constant **CHAR_BIT**. The LIMITS.H file defines **CHAR_BIT** as 8.

Character Sets

Mapping members of the source character set (§3.1.3.4)

The source character set and execution character set include the ASCII characters listed in Table B.1. Escape sequences are also shown in Table B.1.

Table B.1 Escape Sequences

Escape Sequence	Character	ASCII Value
\a	Alert/bell	7
\b	Backspace	8
\f	Formfeed	12

Table B.1 Escape Sequences *(continued)*

Escape Sequence	Character	ASCII Value
\n	Newline	10
\r	Carriage return	13
\t	Horizontal tab	9
\v	Vertical tab	11
\"	Double quotation	34
\ '	Single quotation	39
\\	Backslash	92

Unrepresented Character Constants

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant (§3.1.3.4)

There are no character constants or escape sequences that cannot be represented in the extended character set.

Wide Characters

The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (§3.1.3.4)

The regular character constant, `'ab'` has the integer value (int)0x6162. When there is more than one byte, previously read bytes are shifted left by the value of **CHAR_BIT** and the next byte is compared using the bitwise-OR operator with the low **CHAR_BIT** bits. The number of bytes in the multibyte character constant may not exceed **sizeof(int)**, which is 2 for 16-bit target code, 4 for 32-bit target code.

The multibyte character constant is read as above and this is converted to a wide character constant using the **mbtowc** run-time function. If the result is not a valid wide character constant, an error is issued. In any event, the number of bytes examined by the **mbtowc** function is limited to the value of **MB_CUR_MAX**.

Converting Multibyte Characters

The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (3.1.3.4)

Microsoft C/C++ 7.0 supports only the "C" locale, which does not include any true multibyte characters. In the "C" locale, the **mbtowc** function maps from the ANSI [8859] character set to Unicode.

Range of char Values

Whether a "plain" char has the same range of values as a signed char or an unsigned char (§3.2.1.1)

All character values range from 0x00 to 0xFF, signed or unsigned. If a **char** is not explicitly marked as **signed** or **unsigned**, it defaults to the **signed** type.

The CL option /J changes the default from **signed** to **unsigned**.

B.5 Integers

Range of Integer Values

The representations and sets of values of the various types of integers (§3.1.2.5)

Short integers contain 16 bits (two bytes). Long integers contain 32 bits (four bytes). Signed integers are represented in two's-complement form. The most-significant bit holds the sign: 1 for negative, 0 for positive and zero. The values are listed below:

Type	Minimum and Maximum
unsigned short	0 to 65535
signed short	–32768 to 32767
unsigned long	0 to 4294967295
signed long	–2147483648 to 2147483647

Demotion of Integers

The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (§3.2.1.2)

When a **long** integer is cast to a **short**, or a **short** is cast to a **char**, the least-significant bytes are retained.

For example, this line

```
short x = (short)0x12345678L;
```

assigns the value 0x5678 to `x`, and this line

```
char y = (char)0x1234;
```

assigns the value 0x34 to `y`.

When signed variables are converted to unsigned and vice versa, the bit patterns remain the same. For example, casting –2 (0xFE) to an unsigned value yields 254 (also 0xFE).

Signed Bitwise Operations

The results of bitwise operations on signed integers (§3.3)

Bitwise operations on signed integers work the same as bitwise operations on unsigned integers. For example,`-16 & 99` can be expressed in binary as

```
  11111111 11110000
& 00000000 01100011
  -----------------
  00000000 01100000
```

The result of the bitwise AND is 96.

Remainders

The sign of the remainder on integer division (§3.3.5)

The sign of the remainder is the same as the sign of the dividend. For example,

```
 50 / -6 == -8
 50 % -6 ==  2
-50 /  6 == -8
-50 %  6 == -2
```

Right Shifts

The result of a right shift of a negative-value signed integral type (§3.3.7)

Shifting a negative value to the right yields half the absolute value, rounded down. For example, –253 (binary 11111111 00000011) shifted right one bit produces –127 (binary 11111111 10000001). A *positive* 253 shifts right to produce +126.

Right shifts preserve the sign bit. When a signed integer shifts right, the most-significant bit remains set. When an unsigned integer shifts right, the most-significant bit is cleared. Thus, if 0xF000 is signed, a right shift produces 0xF800. If 0xF000 is unsigned, the result is 0x7800.

Shifting a positive number right sixteen times produces 0x0000. Shifting a negative number right sixteen times produces 0xFFFF.

B.6 Floating-Point Math

Values

The representations and sets of values of the various types of floating-point numbers (§3.1.2.5)

The **float** type contains 32 bits: 1 for the sign, 8 for the exponent, and 23 for the mantissa. Its range is +/– 3.4E38 with at least 7 digits of precision.

The **double** type contains 64 bits: 1 for the sign, 11 for the exponent, and 52 for the mantissa. Its range is +/– 1.7E308 with at least 15 digits of precision.

The **long double** type is new to Version 7.0 of Microsoft C. It contains 80 bits: 1 for the sign, 15 for the exponent, and 64 for the mantissa. Its range is +/– 1.2E4932 with at least 17 digits of precision.

Casting Integers to Floating-Point Values

The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (§3.2.1.3)

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

For example, casting an **unsigned long** (with 32 bits of precision) to a **float** (whose mantissa has 23 bits of precision) rounds the number to the nearest multiple of 256. The **long** values 4,294,966,913 – 4,294,967,167 are all rounded to the **float** value 4,294,967,040.

Truncation of Floating-Point Values

The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (§3.2.1.4)

When an underflow occurs, the value of a floating-point variable is rounded down to zero. An overflow causes a run-time math error.

B.7 Arrays and Pointers

Largest Array Size

The type of integer required to hold the maximum size of an array—that is, the size of size_t (§3.3.3.4, 4.1.1)

The **size_t** typedef is an **unsigned short**, with the range 0x0000 to 0xFFFF. Huge arrays can exceed this limit if they contain more than 65,535 elements. Arithmetic operations on huge arrays should therefore cast **size_t** and the result of an arithmetic operation on pointers to **unsigned long**.

Casting Pointers

The result of casting a pointer to an integer or vice versa (§3.3.4)

Near pointers are the same size as short integers; casting near to short (or short to near) has no immediate effect on the value.

Far pointers and huge pointers are the same size as long integers. Casting far/huge to long (or long to far/huge) has no immediate effect on the value.

When a near pointer is cast to a long, the 16-bit value is "normalized," which means the segment (usually DS) and offset are combined to produce a 32-bit memory location.

When a far or huge pointer is cast to a short, the long value is truncated to a short.

The compiler normalizes based pointers when necessary, unless the based pointer is a constant zero, in which case it is assumed to be a null pointer. See Chapter 12, "Writing Portable C Programs," in the *Programming Techniques* manual for more information about based pointers.

Pointer Subtraction

The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t (§3.3.6, 4.1.1)

A **ptrdiff_t** is a signed integer in the range –32,768 to 32,767, with one exception. Because huge pointers can address more than 64K of memory, subtracting one huge pointer from another can yield a result that is a long integer. The result of subtracting two huge pointers should be cast to a long.

The compiler normalizes based pointers when necessary. In most cases, based pointers are treated as far pointers.

B.8 Registers

Availability of Registers

The extent to which objects can actually be placed in registers by use of the register storage-class specifier (§3.5.1)

Two registers, SI and DI, are available for 16-bit targets with Microsoft C, and ESI, EDI, and EBX are available for 32-bit targets. Register variables with a type that has 16 (or 32) bits may be allocated in these registers.

B.9 Structures, Unions, Enumerations, and Bit Fields

Improper Access to a Union

A member of a union object is accessed using a member of a different type (§3.3.2.3)

If a union of two types is declared and one value is stored, but the union is accessed with the other type, the results are unreliable.

For example, a union of **float** and **int** is declared. A **float** value is stored, but the program later accesses the value as an **int**. In such a situation, the value would depend on the internal storage of **float** values. The integer value would not be reliable.

Padding and Alignment of Structure Members

The padding and alignment of members of structures (§3.5.2.1)

Structure members are aligned to the minimum of their own size or the current packing size. For 16-bit targets, the default packing size is 2. The default corresponds to the /Zp2 command-line option. The default packing size is 4 for 32-bit targets. See page 70, "Storage and Alignment of Structures," for more information.

Sign of Bit Fields

Whether a "plain" int field is treated as a signed int bit field or as an unsigned int bit field (§3.5.2.1)

Bit fields can be signed or unsigned. Plain bit fields are treated as signed.

Storage of Bit Fields

The order of allocation of bit fields within an int (§3.5.2.1)

Bit fields are allocated within a 16-bit integer from least-significant to most-significant bit. In the following code,

```
struct mybitfields
{
   unsigned a : 4;
   unsigned b : 5;
   unsigned c : 7;
} test;

void main( void )
{
   test.a = 2;
   test.b = 31;
   test.c = 0;
}
```

the bits for the integer 0x01F2 would be arranged as follows:

```
00000001 11110010
cccccccb bbbbaaaa
```

Since the 80*x*86 processors store the low byte of integer values before the high byte, the integer 0x01F2 above would be stored in physical memory as 0xF2 followed by 0x01.

Alignment of Bit Fields

Whether a bit field can straddle a storage-unit boundary (§3.5.2.1)

Bit fields default to size **short**, which can cross a byte boundary but not a 16-bit boundary. If the size and location of a bit field would cause it to overflow the current integer, the field is moved to the beginning of the next available integer.

If a bit field is declared as a **long**, it can hold up to 32 bits.

In either case, an individual field cannot cross a 16- or 32-bit boundary.

The enum Type

The integer type chosen to represent the values of an enumeration type (§3.5.2.2)

A variable declared as **enum** is an **int**.

B.10 Qualifiers

Access to Volatile Objects

What constitutes an access to an object that has volatile-qualified type (§3.5.5.3)

Any reference to a volatile-qualified type is an access.

B.11 Declarators

Maximum Number

The maximum number of declarators that can modify an arithmetic, structure, or union type (§3.5.4)

Microsoft C does not limit the number of declarators. The number is limited only by available memory.

B.12 Statements

Limits on Switch Statements

The maximum number of case values in a switch statement (§3.6.4.2)

Microsoft C does not limit the number of **case** values in a **switch** statement. The number is limited only by available memory.

B.13 Preprocessing Directives

Character Constants and Conditional Inclusion

Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value (§3.8.1)

The character set used in preprocessor statements is the same as the execution character set. The preprocessor recognizes negative character values.

Including Bracketed Filenames

The method for locating includable source files (§3.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A "parent" file is the file that has the **#include** directive in it. Instead, it begins by searching for the file in the directories specified on the compiler command line following /I. If the /I option is not present or fails, the preprocessor uses the INCLUDE environment variable to find any include files within angle brackets. The INCLUDE environment variable can contain multiple paths separated by semicolons (;). If more than one directory appears as part of the /I option or within the INCLUDE environment variable, the preprocessor searches them in the order they appear.

See "Include Files" on page 200 for more information.

Including Quoted Filenames

The support for quoted names for includable source files (§3.8.2)

If you specify a complete, unambiguous path specification for the include file between two sets of double quotation marks (“ ”), the preprocessor searches only that path specification and ignores the standard directories.

For include files specified as **#include** "*path-spec*", directory searching begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename were enclosed in angle brackets.

See “Include Files” on page 200 for more information.

Character Sequences

The mapping of source file character sequences (§3.8.2)

Preprocessor statements use the same character set as source file statements with the exception that escape sequences are not supported.

Thus, to specify a path for an include file, use only one backslash:

```
#include "path1\path2\myfile"
```

Within source code, two backslashes are necessary:

```
fil = fopen( "path1\\path2\\myfile", "rt" );
```

Pragmas

The behavior on each recognized #pragma directive (§3.8.6)

The following pragmas are defined for the Microsoft C compiler:

alloc_text	**data_seg**	**linesize**	**pagesize**
auto_inline	**function**	**message**	**skip**
check_pointer	**hdrstop**	**native_caller**	**subtitle**
check_stack	**check_stack**	**optimize**	**title**
code_seg	**inline_recursion**	**pack**	**warning**
comment	**intrinsic**	**page**	

See “Pragma Directives” on page 209 for more information.

Default Date and Time

The definitions for _DATE_ and _TIME_ when, respectively, the date and time of translation are not available (§3.8.8)

When the operating system does not provide the date and time of translation, the default values for _DATE_ and _TIME_ are `May 03 1957` and `17:00:00"`.

B.14 Library Functions

NULL Macro

The null pointer constant to which the macro NULL expands (§4.1.5)

Several include files define the NULL macro as `((void *)0)`.

Diagnostic Printed by the assert Function

The diagnostic printed by and the termination behavior of the assert function (§4.2)

The **assert** function prints a diagnostic message and calls the abort routine if the expression is false (0). The diagnostic message has the form

Assertion failed: [*expression*], file [*filename*], line [*linenumber*]

where *filename* is the name of the source file and *linenumber* is the line number of the assertion that failed in the source file. No action is taken if *expression* is true (nonzero).

Character Testing

The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, and isupper functions (§4.3.1)

Function	Tests For
isalnum	Characters 0–9, A–Z, a–z ASCII 48–57, 65–90, 97–122
isalpha	Characters A–Z, a–z ASCII 65–90, 97–122
iscntrl	ASCII 0–31, 127

Function	Tests For
islower	Characters a–z ASCII 97–122
isprint	Characters A–Z, a–z, 0 – 9, punctuation, space ASCII 32–126
isupper	Characters A–Z ASCII 65–90

Domain Errors

The values returned by the mathematics functions on domain errors (§4.5.1)

The ERRNO.H file defines the domain error constant **EDOM** as 33.

Underflow of Floating-Point Values

Whether the mathematics functions set the integer expression errno to the value of the macro* ERANGE *on underflow range errors (§4.5.1)

A floating-point underflow does not set the expression **errno** to ERANGE. When a value approaches zero and eventually underflows, the value is set to zero.

The fmod Function

Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero (§4.5.6.4)

When the **fmod** function has a second argument of zero, the function returns zero.

The signal Function

The set of signals for the signal function (§4.7.1.1)

The first argument passed to **signal** must be one of the symbolic constants described in the *Run-Time Library Reference* manual for the **signal** function. The information in the *Run-Time Library Reference* also lists the operating mode support for each signal. The constants are also defined in SIGNAL.H.

Default Signals

If the equivalent of signal (sig, SIG_DFL) is not executed prior to the call of a signal handler, the blocking of the signal that is performed (§4.7.1.1)

Signals are set to their default status when a program begins running.

The SIGILL Signal

Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function (§4.7.1.1)

SIGILL is not generated under DOS. It is included for ANSI compatibility. DOS does not provide a way for an application to regain control when an illegal instruction occurs. However, a user can issue a signal function and later trigger that signal via an explicit call to the raise function. As with all signals under DOS, the signal handler is set to the default action (SIG_DFL) before the user's signal handler gets control.

Terminating Newline Characters

Whether the last line of a text stream requires a terminating newline character (§4.9.2)

Stream functions recognize either new line or end of file as the terminating character for a line.

Blank Lines

Whether space characters that are written out to a text stream immediately before a newline character appear when read in (§4.9.2)

Space characters are preserved.

Null Characters

The number of null characters that can be appended to data written to a binary stream (§4.9.2)

Any number of null characters can be appended to a binary stream.

File Position in Append Mode

Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file (§4.9.3)

When a file is opened in append mode, the file position indicator initially points to the end of the file.

Truncation of Text Files

Whether a write on a text stream causes the associated file to be truncated beyond that point (§4.9.3)

Writing to a text stream does not truncate the file beyond that point.

File Buffering

The characteristics of file buffering (§4.9.3)

Disk files accessed through standard I/O functions are fully buffered. By default, the buffer holds 512 bytes. Some of the low-level DOS and BIOS functions (all of which are non-ANSI) are unbuffered.

Zero-Length Files

Whether a zero-length file actually exists (§4.9.3)

Files with a length of zero are permitted.

Filenames

The rules for composing valid file names (§4.9.3)

A file specification can include an optional drive letter (always followed by a colon), a series of optional directory names (separated by backslashes), and a filename.

Filenames and directory names can contain up to eight characters followed by a period and a three-character extension. Case is ignored. The wild-card characters * and ? are not permitted within the name or extension.

File Access Limits

Whether the same file can be open multiple times (§4.9.3)

Opening a file that is already open is not permitted.

Deleting Open Files

The effect of the remove function on an open file (§4.9.4.1)

The **remove** function deletes a file, even if the file is open.

Renaming with a Name That Exists

The effect if a file with the new name exists prior to a call to the rename function (§4.9.4.2)

If you attempt to rename a file using a name that exists, the **rename** function fails and returns an error code.

Printing Pointer Values

The output for %p conversion in the fprintf function (§4.9.6.1)

Microsoft C supports three types of pointer conversions: **%p** (a pointer), **%lp** (a 32-bit far pointer), and **%hp** (a 16-bit near pointer).

The **fprintf** function produces hexadecimal values of the form *XXXX* (an offset) for near pointers or *XXXX:XXXX* (a segment plus an offset, separated by a colon) for far pointers. The output for **%p** depends on the memory model in use.

Reading Pointer Values

The input for %p conversion in the fscanf function (§4.9.6.2)

When the **%p** format character is specified, the **fscanf** function converts pointers from hexadecimal ASCII values into the correct address.

Reading Ranges

The interpretation of a dash (–) character that is neither the first nor the last character in the scanlist for % [conversion in the fscanf function (§4.9.6.2)

The following line

```
fscanf( fileptr, "%[A-Z]", strptr);
```

reads any number of characters in the range A–Z into the string to which `strptr` points.

File Position Errors

The value to which the macro errno is set by the fgetpos or ftell function on failure (§4.9.9.1, 4.9.9.4)

When **fgetpos** or **ftell** fails, **errno** is set to the manifest constant **EINVAL** if the position is invalid or **EBADF** if the file number is bad. The constants are defined in ERRNO.H.

Messages Generated by the perror Function

The messages generated by the perror function (§4.9.10.4)

The **perror** function generates these messages:

```
0  Error 0
1
2  No such file or directory
3
4
5
6
7  Arg list too long
8  Exec format error
9  Bad file number
10
11
12 Not enough core
13 Permission denied
14
15
16
17 File exists
18 Cross-device link
19
20
21
```

```
22 Invalid argument
23
24 Too many open files
25
26
27
28 No space left on device
29
30
31
32
33 Math argument
34 Result too large
35
36 Resource deadlock would occur
```

Allocating Zero Memory

The behavior of the calloc, malloc, or realloc function if the size requested is zero (§4.10.3)

The **calloc**, **malloc**, and **realloc** functions accept zero as an argument. No actual memory is allocated, but a valid pointer is returned and the memory block can be modified later by **realloc**.

The abort Function

The behavior of the abort function with regard to open and temporary files (§4.10.4.1)

The **abort** function does not close files that are open or temporary. It does not flush stream buffers.

The atexit Function

The status returned by the atexit function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE (§4.10.4.3)

The **atexit** function returns zero if successful, or a nonzero value if unsuccessful.

Environment Names

The set of environment names and the method for altering the environment list used by the getenv function (§4.10.4.4)

The set of environment names is unlimited.

To change environment variables from within a C program, call the **putenv** function. To change environment variables from the DOS command line, use the SET command (for example, SET LIB = D:\ LIBS).

Environment variables exist only as long as their host copy of DOS is running. For example, the line

```
system( "SET LIB = D:\LIBS" );
```

would run a copy of DOS, set the environment variable LIB, and return to the C program, exiting the secondary copy of DOS. Exiting that copy of DOS removes the temporary environment variable LIB.

Likewise, changes made by the **putenv** function last only until the program ends.

The system Function

The contents and mode of execution of the string by the system function (§4.10.4.5)

The **system** function executes an internal DOS command, or an EXE, COM, or BAT file from within a C program rather than from the command line.

It examines the COMSPEC environment variable to find the command interpreter, which is typically COMMAND.COM in DOS. The **system** function then passes the argument string to the command interpreter.

The strerror Function

The contents of the error message strings returned by the strerror function (§4.11.6.2)

The **strerror** function generates these messages:

```
0   Error 0
1
2   No such file or directory
3
4
5
6
7   Arg list too long
8   Exec format error
9   Bad file number
10
11
12  Not enough core
13  Permission denied
```

```
14
15
16
17  File exists
18  Cross-device link
19
20
21
22  Invalid argument
23
24  Too many open files
25
26
27
28  No space left on device
29
30
31
32
33  Math argument
34  Result too large
35
36  Resource deadlock would occur
```

The Time Zone

The local time zone and Daylight Saving Time (§4.12.1)

The local time zone is Pacific Standard Time. Microsoft C supports Daylight Saving Time.

The clock Function

The era for the clock function (§4.12.2.1)

The **clock** function's era begins (with a value of 0) when the C program starts to execute. It returns times measured in 1/CLOCKS_PER_SEC (which equals 1/1000 for Microsoft C version 7.0).

Appendix C

Differences Between C Versions 6.0 and 7.0

This appendix describes the differences between versions 6.0 and 7.0 of Microsoft C, including additions, deletions, and changes. Some of the changes are required by the American National Standards Institute (ANSI) standard for the C programming language. Other changes improve or augment the existing capabilities of the compiler.

Many of the changes will have no effect on code that was written and compiled with previous versions of Microsoft C. In some cases, however, you may have to modify or correct existing code before compiling with version 7.0.

C.1 New Features

The features described in this section are new to Microsoft C/C++ version 7.0.

Support for C++

The C++ compiler provided with version 7.0 implements the C++ programming language as described in the *Annotated C++ Reference* by Margaret A. Ellis and Bjarne Stroustrup. See the *C++ Tutorial* and the *C++ Language Reference* for more information.

Precompiled Header Files

Precompiled header files reduce compilation time for both C and C++ programs. See Chapter 2 in *Programming Techniques* for information on creating and using precompiled headers.

P-Code

Compiling programs or parts of programs into p-code typically reduces code size by a factor of four. This is a useful technique when code size is more important than speed. See Chapter 3 in *Programming Techniques* for information on p-code.

New Pragmas

New pragmas in Microsoft C, version 7.0, include **auto_inline**, **code_seg**, **data_seg**, **hdrstop**, **inline_depth**, **inline_recursion**, **native_caller**, and **warning**. See page 209, "Pragma Directives," for information on these pragmas.

Inline Functions

The **__inline** keyword tells the compiler that it can substitute the code within the function definition for every instance of a function call. Substitution occurs at the discretion of the compiler. You can specify that individual functions be **__inline** functions or you can use the /Ob2 command-line option to have the compiler optimize your code by generating as many functions inline as possible. The **__inline** keyword (and the **inline** keyword when compiling with /Ze) are new keywords for Microsoft C, version 7.0. See page 173 for more information.

New Intrinsic Functions

The command-line option for intrinsic function optimization (/Oi) or the pragma **intrinsic** causes the compiler to generate inline code for functions. Microsoft C version 7.0 adds many intrinsic functions. See "Pragma Directives" on page 209 for a list of all intrinsic functions.

Function Allocation Using __based

Due to bug fixes since version 6.0 of the compiler, programs that use **__based** incorrectly may not compile with Microsoft C, version 7.0.

Only lvalues may be converted to **__based((__segment)__self)** pointers. Since the right side of an assignment gets converted to the type of the left hand side, this means you must now use an explicit cast when assigning to a **__based((__segment)__self)** pointer. For example,

```
int __based((__segment)__self) *piself;
piself = 1;
```

is interpreted as

```
piself = (int __based((__segment)__self)) 1;  /* Illegal! */
```

but the expression 1 does not have a segment so this is illegal. To put the offset 1 into this based pointer, you must cast it to a **__based(void)** pointer as shown:

```
piself = (int --based(void)*) 1;
```

Similarly,

```
unsigned short us;
piself = ui;
```

must be written as

```
piself = (int __based(void)*) ui;
```

if you want to move the contents of ui into the **__based((__segment)__self)** pointer.

Also new to Microsoft C, version 7.0, functions can now be declared as based on a segment constant if a function needs to be allocated in a given segment. This feature replaces the **alloc_text** pragma.

When a segment name ends with _TEXT, the compiler converts it to a code segment rather than a data segment. For example, for this declaration,

```
void funct1()
{
    static char _based(_segname("MY_TEXT")) arr[] = "A string\n";
}
```

the C version 6.0 compiler made MY_TEXT a data segment, but the C 7.0 compiler makes MY_TEXT a code segment.

See Chapter 4 of *Programming Techniques* for more information.

New Run-Time Library Functions

Microsoft C/C++ provides several new functions to support virtual memory. See Chapter 2 in the *Run-Time Library Reference* for a list. A list of new functions provided to support Microsoft QuickWin is also in Chapter 2 of that manual.

In addition to the **_osmajor**, **_osminor, _osversion**, and **_osmode** variables, Microsoft C/C++ now provides the **_cpumode** variable. It returns either **_REAL_MODE** or **_PROTECT_MODE** to indicate the mode of the current process.

The new **_fatexit** and **_fonexit** functions provide model-independent processing at program termination.

The new library functions **_snprintf** and **_vsnprintf** can be used to control the size of a formatted string written to a buffer.

Microsoft C/C++ now provides the ability to bypass the operating system buffers and flush a file directly to disk. The new functions that provide this capability are **_commit** in IO.H and **_dos_commit** in DOS.H. The **fdopen** and **fopen** functions have new flags to specify either mode. To take advantage of this with applications written prior to Microsoft C, version 7.0, you can link your application with COMMODE.OBJ in your \C700\LIB directory.

See the *Run-Time Library Reference* for complete information on all these new library functions.

New CL Command-Line Options

The following CL command-line options are new to Microsoft C 7:

Option	**Action**
/Bm*memavailable*	Sets the amount of memory available to the compiler.
/f	Specify fast compile (replaces /qc).
/Fp*filename*	Specify precompiled header filename.
/GA	Optimize entry/exit code for protected-mode Windows applications.
/GD	Optimize entry/exit code for protected-mode Windows DLLs.
/GE*string*	Optimize entry/exit code for protected-mode Windows DLLs.
/Gn	Remove p-code native entry points.
/Gp*number*	Specify maximum number of entry tables.
/Gq	Specify real mode Windows (for compatibility with /Gw).
/Gx	Assume all data is near.
/Gy	Enable function-level linking.
/Ld	Control library selection for DLLs.
/Lw	Control library selection for applications.
/Lu	Link without C run-time startup code.
/NQ*pcodesegment*	Name temporary segment for p-code.
/NV	Name temporary segment for far virtual tables.
/Ob*number*	Control inline expansion.
/Oq	Turn on p-code optimization.

Option	Action
/Of[[-]]	Turn on (or off) p-code quoting.
/Oo[[-]]	Turn on (or off) post code-generation optimizing.
/Ov[[-]]	Sort local variables by frequency of use (or in the order they appear) for p-code.
/Tp	Specify C++ source file.
/Yc[[*filename*]]	Create precompiled header.
/Yd	Include debugging information in precompiled header file.
/Yu[[*filename*]]	Use precompiled header file.
/Zf	Accept **__far** keyword (32-bit code only)
/Zn	Turn off SBRPACK utility.

Support for Super VGA Screen Modes

Microsoft C/C++ provides eight new manifest constants that support the Super VGA screen modes specified by the Video Electronic Standards Association (VESA). These are **_ORES256COLOR**, **_VRES256COLOR**, **_SRES16COLOR**, **_SRES256COLOR**, **_XRES16COLOR**, **_XRES256COLOR**, **_ZRES16COLOR**, and **_ZRES256COLOR**. Other non-standard Super VGA nodes may also be supported; see the *Programming Techniques* manual for important warranty information.

C.2 Changes and Deletions

A number of changes have been made to the compiler to support the ANSI standard. By default, the Microsoft-extensions to the compiler are enabled (/Ze). When you compile with /Za (disable Microsoft extensions), the compiler generates errors and warnings for code that does not conform to the ANSI standard.

Note There are no areas of nonconformance to the ANSI standard in Microsoft C version 7.0, when compiling with the /Za command-line option.

The changes and deletions listed in this section may affect existing programs.

ANSI-Mandated New Features

The following ANSI-mandated features are new to version 7.0:

- Wide characters (type (**wchar_t**) have been implemented. See page 8 for more information.

- Complete parameter declarations (`int a`) and abstract declarations (`int`) are allowed in the same declaration.
- The STDLIB.H header file contains five new functions (**mblen**, **mbstowcs**, **mbtowc**, **wcstombs**, and **wctomb**), and a new macro, **MB_CUR_MAX**, for wide character support.

  Microsoft C version 7.0, also provides model-independent versions of these new functions (**_fmblen**, **_fmbtowc**, **_fwctomb**, **_fmbstowcs**, and **_fwctombs**), although these functions are not mandated by ANSI.

Return Statements Containing Expressions

A **return** statement with an expression in a function returning **void** now generates a level 1 warning and the code in the **return** expression is not evaluated. In this example with Microsoft C version 6.0, `func1` was called, but `func1` is not called when using Microsoft C version 7.0.

```
void funct1()
{
}

void funct2()
{
    return funct1();
}
```

Function Declarations

To comply with the ANSI standard, old-style function declarations using an ellipsis now generate an error when compiling with /Za and a level 4 warning when compiling with /Ze. For example,

```
void funct1( a, ... )  /* Generates a warning under /Ze or */
int a;                 /* an error when compiling with /Za */
{
}
```

You should rewrite this declaration to as a prototype:

```
void funct1( int a, ... )
{
}
```

See page 166 for information on old-style function declarations.

Old-style function declarations also generate warnings if you subsequently declare or define the same function with either an ellipsis or a parameter with a type that is not the same as its promoted type.

Type Checking

Type checking is now ANSI-compliant which means that type **short** and type **int** are distinct types. For example, this is a redefinition in Microsoft C 7.0 that was accepted by version 6.0.

```
int   myfunc();
short myfunc();
```

This next example also generates a warning about indirection to different types:

```
int *pi;
short *ps;

ps = pi;  /* Generates warning under C 7.0 */
```

The C 7.0 compiler also generates warnings for differences in sign. For example,

```
signed int *pi;
unsigned int *pu

pi = pu;  /* Generates warning under C 7.0 */
```

Prototype Scope

Prototype scope is now ANSI-compliant when compiling with the /Za command-line option. This means that if you declare a **struct** or **union** tag within a prototype, the tag is entered at that scope rather than at global scope. For example, under ANSI you can never call this function without getting a type mismatch error:

```
void func1( struct S * );
```

To correct your code, define or declare the **struct** or **union** at global scope before the function prototype:

```
struct S;
void func1( struct S * );
```

Under /Ze, the tag is still entered at global scope.

Naming Conventions

ANSI specifies that identifiers that begin with two underscores are reserved in all scopes for use by the implementation. In Microsoft C 7.0, you should avoid using your own identifiers with these names because they may conflict with existing or future Microsoft-specific identifiers.

ANSI also specifies that identifiers that begin with a single underscore and a lower case letter are reserved at file scope for use by the implementation. This means that you should not start your own global variable and function names with an underscore because they may conflict with existing or future MS-specific identifiers.

Under /Ze, Microsoft C 7.0 still permits references to old Microsoft-specific identifier names that did not follow the ANSI rules for implementation-reserved identifiers. For example, you may use either **_MSDOS** or **MSDOS** to test whether you are compiling for MS-DOS, **near** or **__near** to specify a near address, and either **read** or **_read** to call that runtime library function. If you reference MS-specific identifiers that are global variable names or function names and you use the non-ANSI form, you must link with OLDNAMES.LIB. (Under /Ze, your program is automatically linked with OLDNAMES.LIB.)

See "Keywords" on page 4 for the ANSI and Microsoft-specific keywords.

Time Returned by the time Function

The **time** run-time library function now returns the number of seconds elapsed since midnight, December 31, 1899, Coordinated Universal Time, instead of the number of seconds that have elapsed since Greenwich Mean Time, January 1, 1970. This change conforms with the ANSI standard.

Nesting Level for Include Files

The nesting level for include files is now limited only by available memory.

const and volatile in Declarations

The **const** and **volatile** attributes cannot be repeated in a declaration, even through a **typedef**.

```
typedef const int CI;
const CI i;           /* Illegal */
```

Conditional Operator

Typing for the conditional operator (**? :**) now conforms to the ANSI standard. See page 136 for information on the conditional operator.

Visibility of Functions

In C, version 6.0 and in C version 7.0 when compiling with the /Ze command-line option, functions declared within a block using the **extern** keyword have global visibility. This is not true when compiling with /Za. This feature should not be relied upon if portability of source code is a consideration.

Macro Redefinition

When you compile with /Za (Microsoft extensions disabled), the compiler generates a warning if two macros are the same except for the spelling of a macro argument. The C 7.0 compiler generates a warning when it encounters the second macro in this example:

```
#define findnum( a ) a
#define findnum( b ) b
```

Version 6.0 of the Microsoft C compiler did not generate a warning in this case. The macro expansion is the same.

Unary Arithmetic Operators on Pointers

Unary arithmetic operators on pointers are now illegal. For example, this code generates an error:

```
char *p;
func1( -p );        /* Illegal */
```

New Errors and Warnings

The C 7.0 compiler catches more problems and therefore generates more errors and warnings than the C 6.0 compiler. Warning levels are new adjustable, and you can also specify that warnings display only once or not at all. See "Pragma Directives" on page 209 for information on pragma **warning**.

- Unrecognized escape sequences now generate a level 1 warning instead of level 4 the backslash is an escape sequence in strings and some operating systems use the backslash as a path separator.
- The compiler generates a level 1 warning if a header file uses **#pragma pack** to change the packing size for a structure and does not reset the packing size to the original level.

The following changes to errors and warnings were made in order to conform with the ANSI standard:

- A hex escape sequence used as a **char** constant generates an error if it exceeds the range of type **char** (or the range of wide characters for type **whar_t**).
- Overflow on constant expressions now generate warnings.
- Declarations that do not declare at least one declarator, tag, or enum member now generate a warning.
- Translation units that do not contain at least one external declaration generate warnings.
- Attempting to take the address of a **register** array, either explicitly or implicitly, now generates an error.
- Function parameters declared with the **auto** attribute generate an error.
- Block-scope function declarations with a storage-class specifier other than **extern** generate errors.
- When compiling with /Za, the **main** function must conform to either of the following or the compiler generates an error:

```
int main( void )

int main( int argc, char *argv[] )
```

- Escaped newline characters in single-line comments (comments preceded by //) now generate a level 1 warning.

Changes to Calling Conventions

The **__syscall** and **__stdcall** calling conventions are not supported for 16_bit targets. For 32-bit targets, the **__pascal**, **__fortran**, and **__syscall** calling conventions are not supported, but **__stdcall** is supported. See page 169, “Specifying Calling Conventions,” for more information.

Expanded Functionality with __export

You can use the /GA and /GD command-line options with **__export** to selectively optimize the entry or exit code for protected mode Windows applications and dynamic-link libraries.

Obsolete Pragmas

The **same_seg** pragma is no longer supported. Use the **__based** keyword instead. The **loop_opt** pragma has been replaced with the **optimize** pragma.

See page 209, “Pragma Directives,” for more information.

Obsolete and Changed Command-Line Options

The /MD, /ML, /MT, /Lp, /Lc, /Li, /Gi, /B1, /B2, and /B3 command-line options are no longer supported.

The behavior of /u has changed. In Microsoft C version 6.0, the /u option removed definitions of all predefined identifiers. In version 7.0, the action of the /u option has been expanded to turn off every defined identifier.

The /D command-line option has expanded functionality in Microsoft C version 7.0. The /D option now accepts a pound sign (#) as an alternative to the equal sign (=). This allows you to use the CL environment variable to set precompiler macros as follows:

```
SET CL="/DQUOTES#1"
```

See Chapter 13 in *Environment and Tools* for more information on CL command-line options.

Linking Considerations

Object modules produced by the C 7.0 compiler cannot be linked with the C 6.0 linker. See Chapter 14 in *Environment and Tools* for more information on new linker features.

Changes to Constants

The value of **MB_LEN_MAX** has changed from 1 to 2 to conform to the ANSI standard.

Alternate Math Library

The alternate math library does not support type **long double**.

Obsolete Functions

The following OS/2-specific functions are not supported by Microsoft C version 7.0: **_beginthread, _cwait, _endthread, _pclose, _pipe, _popen, _wait.**

Storage of Strings

The compiler does not guarantee that identical strings will be stored at different addresses. Code for Microsoft C, version 7.0, should not depend on identical addresses for identical strings.

Index

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®

1191 Part No. 24771