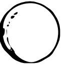
C Compiler Reference

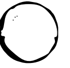
IBM Personal Computer XENIX™ Software Development System

Programming Family

Personal Computer Software

6138701





C Compiler Reference

IBM Personal Computer XENIX™ Software Development System

Programming Family

Personal
Computer
Software

First Edition (December 1984)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Products are not stocked at the address below. Requests for copies of this publication and for technical information about IBM Personal Computer products should be made to your authorized IBM Personal Computer dealer or your IBM Marketing Representative.

The following paragraph applies only to the United States and Puerto Rico: A Reader's Comment Form is provided at the back of this publication. If the form has been removed, address comments to: IBM Corporation, Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

© Copyright International Business Machines Corporation 1984
© Copyright Microsoft Corporation 1983, 1984

IBM Personal Computer XENIX Library Overview

The XENIX[1] System has three available products. They are the:

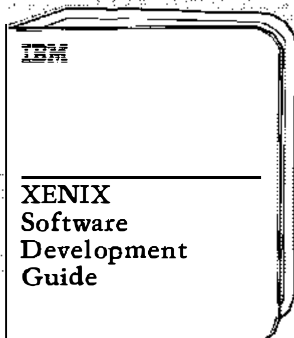
- Operating System
- Software Development System
- Text Formatting System

The following pages outline the XENIX Software Development System library.

[1] XENIX is a trademark of Microsoft Corporation.

XENIX Software Development System

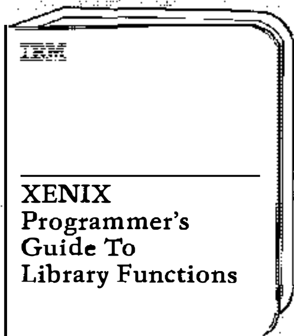
For C Language Users



- Creating language programs
- Invoking the C Compiler
- Program checkers, maintainers, and debuggers
- Using S-Files
- The C-Shell

A guide to the available programming tools in the XENIX environment.

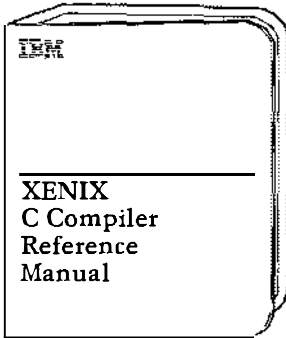
For All Users



- Using stream functions
- Screen processing
- Process controls
- Creating and using pipes
- Using signals and system resources

A reference to system calls, subroutines, and file formats. Use with the XENIX Software Command Reference.

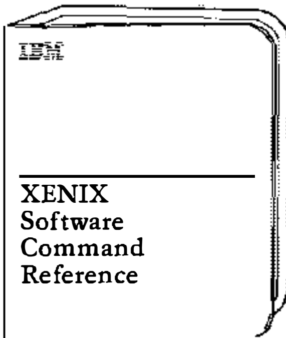
For Experienced Language Users



- Elements of the C programming language
- Preprocessor Directives
- Declarations
- Expressions and Assignments
- Description of functions and statements

A reference to the C programming language. Notational conventions are described throughout the manual.

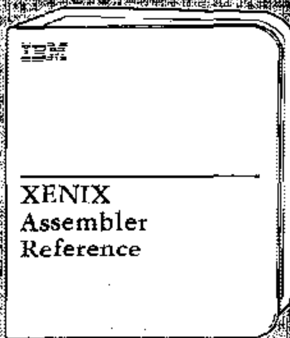
For All Users



- Software Development commands (CP)
- Command definition and syntax
- System calls and subroutines (S)
- System call and library function cross reference

A reference to Software Development System commands. Describes system services in the Operating System kernel.

For Assembler Users



- Assembly Language format
- Operands and Operators
- Directives
- Segment usage
- Machine instructions
- Assembler messages

A reference for programmers who use the IBM Personal Computer XENIX Assembler.

About This Book

This reference manual describes the C language. It is intended to be a reference for programmers with experience in C or in another language. You should have knowledge of programming fundamentals.

This book is organized as follows:

Chapter 1. Elements of C
Describes the letters, numbers and symbols you can use in a C program. Also outlines the combinations of characters having special meanings to the C compiler.

Chapter 2. Program Structure
Discusses the components and structure of C programs. Explains how C source files are organized.

Chapter 3. Declarations
Discusses C declarations, which specify the attributes of C variables, functions and user-defined types.

Chapter 4. Expressions and Assignments
Describes the operands and operators that make up C expressions and assignments. Also discussed are the type conversions and side effects that can accompany the evaluation of expressions.

Chapter 5. Statements
Describes C statements. Shows how statements control the flow of program execution.

Chapter 6. Functions
Discusses the features of C functions; the form of a C function, formal and actual parameters, and return values.

Chapter 7. Preprocessor Directives
Describes the instructions recognized by the C preprocessor. The C preprocessor is a text processor that is automatically invoked before compilation.

Related IBM Personal Computer XENIX Publications

- IBM Personal Computer XENIX Software Development Guide
- IBM Personal Computer XENIX Programmer's Guide to Library Functions
- IBM Personal Computer XENIX Software Command Reference
- IBM Personal Computer XENIX Assembler Reference
- IBM Personal Computer XENIX Installation Guide
- IBM Personal Computer XENIX Visual Shell
- IBM Personal Computer XENIX System Administration
- IBM Personal Computer XENIX Basic Operations Guide
- IBM Personal Computer XENIX Command Reference

Contents

Chapter 1. Elements of C **1-1**
- Introduction 1-3
- Notational conventions 1-3
- The Character Set 1-5
  - Letters and Digits 1-5
  - Whitespace Characters 1-5
  - Punctuation and Special Characters 1-6
  - Escape Sequences 1-7
  - Operators 1-8
- Constants 1-9
  - Integer Constants 1-9
  - Floating-Point Constants 1-11
  - Character Constants 1-12
  - String Constant 1-12
- Identifiers 1-14
- Keywords 1-15
- Comments 1-15
- Tokens 1-17

Chapter 2. Program Structure **2-1**
- Introduction 2-3
- Source Program 2-3
- Source Files 2-5
- Program Execution 2-7
- Directives 2-9
- Scope and Visibility 2-10

Chapter 3. Declarations **3-1**
- Introduction 3-3
- Storage Class Specifiers 3-4
  - Automatic Class 3-4
  - Register Class 3-5
  - Static Class 3-5
  - External Class 3-6
- Type Specifiers 3-7
  - Fundamental Types 3-8
  - Enumeration Types 3-9

Structure Types 3-10
Union Types 3-12
Declarators 3-13
Pointer Modifier 3-14
Array Modifier 3-15
Function Modifier 3-17
Complex Declarators 3-17
Type Declarations 3-20
Enumeration Declarations 3-21
Structure Declarations 3-22
Union Declarations 3-24
Variable Declarations 3-26
Simple Variable Declarations 3-27
Array Declarations 3-28
Structure Declarations 3-30
Union Declarations 3-31
Pointer Declarations 3-33
Function Declarations 3-35
Typedef Declarations 3-36
Initialization 3-38
Fundamental Types 3-38
Pointer Types 3-39
Aggregate Types 3-40
String initializers 3-43
Visibility and Scope 3-44
Global and External Variables 3-45
Static Variables 3-46
Global and External Functions 3-47
Static Functions 3-47
Type names 3-48

Chapter 4. Expressions and Assignments 4-1
Introduction 4-5
Operands 4-6
Constants 4-6
Identifiers 4-6
Integral and floating point identifiers 4-6
Enumeration identifiers 4-6
Structure and union identifiers 4-7
Pointer identifiers 4-7
Array identifiers 4-7
Function identifiers 4-7
Strings 4-8

Function calls 4-9
Subscript expressions 4-10
Member Selection Expressions 4-13
Expressions 4-14
    Expressions with Operators 4-15
    Type Cast Expressions 4-15
    Expressions in Parentheses 4-16
    Constant Expressions 4-16
Type Conversions 4-17
Assignment Conversions 4-18
    Conversions from Signed Integer Types 4-18
    Conversions from Unsigned Integer Types 4-20
    Conversions from Floating Point Types 4-22
    Conversions from Enumeration Type 4-24
    Conversions from Structure and Union Types . . 4-24
    Conversions from pointer types 4-24
    Conversions from Void Type 4-24
Type Cast Conversions 4-25
Operator Conversions 4-25
Function Call Conversions 4-26
Operators 4-27
Complement Operators 4-28
    Arithmetic negation (-) 4-28
    Bitwise complement (~) 4-28
    Logical not (!) 4-28
Indirection and address of operators 4-29
    Indirection (*) 4-29
    Address of (&) 4-29
The sizeof Operator 4-30
Multiplicative Operators 4-31
    Multiplication (*) 4-31
    Division (/) 4-31
    Remainder (%) 4-31
Additive Operators 4-32
    Addition (+) 4-32
    Subtraction (-) 4-33
    Pointer and integer combinations 4-33
    Overflow 4-35
Shift operators 4-35
Relational operators 4-36
Bitwise Operators 4-37
    Bitwise AND (&) 4-37
    Bitwise Inclusive OR (|) 4-37

Bitwise Exclusive OR (∧) 4-37
Logical Operators 4-38
Logical AND (&&) 4-38
Logical OR (| |) 4-39
Sequential Evaluation Operator (,) 4-40
The Conditional Operator (?:) 4-41
Assignment Operators 4-42
Unary Increment and Decrement 4-44
Simple Assignment 4-45
Compound Assignment 4-46
Precedence 4-47
Side Effects 4-51

Chapter 5. Statements 5-1
Introduction 5-3
Break statement 5-5
Syntax 5-5
Execution 5-5
Example: 5-5
Exiting from Nested Statements 5-6
Compound statement 5-7
Syntax 5-7
Execution 5-7
Example: 5-8
Labeling Statements 5-8
Continue statement 5-9
Syntax 5-9
Execution 5-9
Example: 5-9
Do statement 5-10
Syntax 5-10
Execution 5-10
Example: 5-10
Expression statement 5-11
Syntax 5-11
Execution 5-11
Example: 5-11
Assignments and Function Calls 5-11
For statement 5-12
Syntax 5-12
Execution 5-12
Example: 5-13
Goto and labeled statements 5-14

Syntax 5-14
Execution 5-14
Example: 5-14
Forming labels 5-15
If statement 5-15
Syntax 5-15
Execution 5-15
Example: 5-15
Nesting 5-16
Null statement 5-17
Syntax 5-17
Execution 5-17
Example: 5-17
Labeling a null statement 5-17
Return statement 5-18
Syntax 5-18
Execution 5-18
Example: 5-18
Omitting the return statement 5-19
Switch statement 5-20
Syntax 5-20
Execution 5-20
Examples: 5-22
Labeling statements 5-23
While statement 5-24
Syntax 5-24
Execution 5-24
Example: 5-24

Chapter 6. Functions 6-1
Introduction 6-3
Function Definition 6-4
Return Value Type 6-5
Formal Parameters 6-8
Function Body 6-10
Function Declarations 6-11
Static Functions 6-13
Function Calls 6-14
Actual Parameters 6-15
Fundamental Types 6-16
Arrays 6-17
Structures and Unions 6-18
Pointers 6-19

Function Pointers 6-21
Recursive Functions 6-23

Chapter 7. Preprocessor Directives **7-1**
Introduction 7-3
Define Directive 7-4
Undefine Directive 7-7
Include Directive 7-8
If, Elif, Else, and Endif Directives 7-9
If defined and Elif defined Directives 7-12
Ifdef and Ifndef Directives 7-14
Line Control Directive 7-15

Appendix A. Differences **A-3**

Appendix B. C Compiler Messages and Limits **B-1**
Introduction B-1
Compiler Error Messages B-1
Warning Messages B-1
Program Error Messages B-7
Fatal Error Messages B-21
Compiler Limits B-23

Index **Index-1**

Chapter 1. Elements of C

Contents

Introduction **1-3**

Notational conventions **1-3**

The Character Set **1-5**
- Letters and Digits 1-5
- Whitespace Characters 1-5
- Punctuation and Special Characters 1-6
- Escape Sequences 1-7
- Operators 1-8

Constants **1-9**
- Integer Constants 1-9
- Floating-Point Constants 1-11
- Character Constants 1-12
- String Constant 1-12

Identifiers **1-14**

Keywords **1-15**

Comments **1-15**

Tokens **1-17**

Introduction

This chapter describes the elements of the C programming language. The elements of the language are the names, numbers, and characters used to construct the statements and declarations of a C program.

Notational conventions

The following notational conventions are used throughout this manual.

boldface	Boldface marks keywords in the text or in syntax specifications. Keywords in examples are not boldface.
italics	Italics are used in syntax specifications and in the text for general terms marking the places where specific terms will appear in an actual C program. For example, in: goto *name*; *name* is italicized to indicate that this is a general form for the **goto** statement. Italics may also be used for emphasis of particular words in the text.
[ ]	Brackets enclose optional portions in syntax specifications. However, the C language also uses brackets for array declarations and subscript expressions. In discussions of the syntax for array declarations and subscript expressions, and in examples, brackets have the meaning specified by C.

Thus,

```
return [expression];
```

indicates that *expression* can be omitted, while

```
return (var[1]);
```

is an example indicating that the value of the subscript expression var[1] is to be returned.

ellipses . . .

Ellipses following an item indicate that more items having the same form can appear. Ellipses can be vertical or horizontal. For instance,

= {*expression* [,*expression* . . .]}

indicates that one or more expressions separated by commas can appear between the braces, while

```
{
 [declaration]
     .
     .
     .
     statement
    [statement]
     .
     .
     .
}
```

indicates that zero or more declarations, followed by one or more statements, can appear between the braces.

" "

Quotation marks set off values and program fragments within the text. Some C constructs require quotation marks.

The Character Set

C programs are constructed by combining the characters of the C character set into meaningful statements. The C character set consists of letters, digits, and punctuation marks. Each character has an explicit meaning and can only be used as defined. The C compiler generates error messages if a character is not used correctly or if characters that do not belong to the C character set are used.

The following sections describe the characters and symbols of the standard C character set and explain how and when to use them.

Letters and Digits

The C character set contains the uppercase and lowercase letters of the English alphabet and the ten decimal digits of the arabic number system (0 through 9).

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

The C compiler treats uppercase and lowercase letters as distinct characters. If a lowercase *a* is specified in a given item, you cannot substitute an uppercase *A* in its place; you must use the lowercase letter.

Whitespace Characters

Space, tab, line feed, carriage return, form feed, vertical tab, and newline characters are called whitespace characters because they serve the same purpose as the spaces between words and lines on a printed page. These characters separate user-defined items, such as constants and identifiers, from other items within a program, allowing the C compiler to distinguish between them. A Ctrl-Z character (sometimes used for an end-of-file mark) is also a whitespace character.

The C compiler typically ignores a whitespace character unless it is used as a separator or as part of a constant. This means you

can use extra whitespace characters to make a program more readable. Comments, described later in this chapter, are also treated as whitespace.

Punctuation and Special Characters

The punctuation and special characters in the C character set are used for a variety of purposes, from organizing the text of a program to defining the tasks to be carried out by the compiler or by the compiled program. The following table lists these characters.

<table>
<tr><th>Character</th><th>Name</th><th>Character</th><th>Name</th></tr>
<tr><td>,</td><td>comma</td><td>$</td><td>dollar sign</td></tr>
<tr><td>.</td><td>period</td><td>|</td><td>vertical bar</td></tr>
<tr><td>;</td><td>semicolon</td><td>/</td><td>slash</td></tr>
<tr><td>:</td><td>colon</td><td>\</td><td>backslash</td></tr>
<tr><td>?</td><td>question mark</td><td>~</td><td>tilde</td></tr>
<tr><td>'</td><td>single quotation</td><td>_</td><td>underscore</td></tr>
<tr><td>"</td><td>double quotation</td><td>#</td><td>number sign</td></tr>
<tr><td>(</td><td>left parenthesis</td><td>%</td><td>percent sign</td></tr>
<tr><td>)</td><td>right parenthesis</td><td>&</td><td>ampersand</td></tr>
<tr><td>[</td><td>left bracket</td><td>^</td><td>caret</td></tr>
<tr><td>]</td><td>right bracket</td><td>*</td><td>asterisk</td></tr>
<tr><td>{</td><td>left brace</td><td>-</td><td>hyphen; minus sign</td></tr>
<tr><td>}</td><td>right brace</td><td>=</td><td>equal sign</td></tr>
<tr><td><</td><td>left angle bracket</td><td>+</td><td>plus sign</td></tr>
<tr><td>></td><td>right angle bracket</td><td>@</td><td>at sign</td></tr>
<tr><td>!</td><td>exclamation mark</td><td>`</td><td>accent</td></tr>
</table>

These characters have special meaning to the C compiler and can only be used as described in later sections of this manual.

Escape Sequences

Escape sequences are special character combinations that represent whitespace and non-graphic characters. Escape sequences are used when creating strings for display on terminals and printers. A sequence consists of a backslash followed by a letter or combination of digits. The following table lists the C language escape sequences.

Non-graphic Characters for Constants

Escape Sequence	Name
\n	newline
\t	horizontal tab
\v	vertical tab
\b	backspace
\r	carriage return
\f	form feed
\'	single quote
\"	double quote
\\	backslash
\ddd	ASCII character in octal
\xdd	ASCII character in hexadecimal

The sequences \ddd and \xdd, where d represents a digit, allow any character in the ASCII character set to be given as a 3-digit octal or a 2-digit hexadecimal character code. For example, the backspace character can be given as \010 or \x08 . The ASCII null character can be given as \0 or \x0 . Only octal digits may appear in an octal escape sequence.

These sequences allow non-graphic control characters to be sent to a display device. For example, the escape character, \033 , is often used as the first character of a control command for a terminal or printer.

Non-graphic characters should always be represented by escape sequences. Placing a non-graphic character in a C program usually has unexpected results.

Operators

Operators are special character combinations that specify arithmetic, logical, and assignment operations. They are used when forming expressions. The following tables list all the operators of the C language.

Arithmetic and Logical Operators

<table>
<tr><th>Operator</th><th>Name</th><th>Operator</th><th>Name</th></tr>
<tr><td>!</td><td>logical negation</td><td><</td><td>less than</td></tr>
<tr><td>~</td><td>bitwise negation</td><td><=</td><td>less than or equal</td></tr>
<tr><td>++</td><td>increment</td><td>></td><td>greater than</td></tr>
<tr><td>--</td><td>decrement</td><td>>=</td><td>greater than or equal</td></tr>
<tr><td>+</td><td>addition</td><td>==</td><td>equal</td></tr>
<tr><td>-</td><td>subtraction, negative</td><td>!=</td><td>not equal</td></tr>
<tr><td>*</td><td>multiplication, indirection</td><td>|</td><td>bitwise OR</td></tr>
<tr><td>/</td><td>division</td><td>&</td><td>bitwise AND, address</td></tr>
<tr><td>%</td><td>remainder</td><td>^</td><td>bitwise exclusive OR</td></tr>
<tr><td><<</td><td>shift left</td><td>&&</td><td>logical AND</td></tr>
<tr><td>>></td><td>shift right</td><td>||</td><td>logical OR</td></tr>
</table>

Assignment Operators

<table>
<tr><th>Operator</th><th>Name</th></tr>
<tr><td>=</td><td>assign</td></tr>
<tr><td>+=</td><td>increment and assign</td></tr>
<tr><td>-=</td><td>decrement and assign</td></tr>
<tr><td>*=</td><td>multiply and assign</td></tr>
<tr><td>/=</td><td>divide and assign</td></tr>
<tr><td>%=</td><td>modulus and assign</td></tr>
<tr><td>>>=</td><td>shift right and assign</td></tr>
<tr><td><<=</td><td>shift left and assign</td></tr>
<tr><td>&=</td><td>bitwise AND and assign</td></tr>
<tr><td>^=</td><td>bitwise exclusive OR and assign</td></tr>
<tr><td>|=</td><td>bitwise OR and assign</td></tr>
</table>

See Chapter 4, "Expressions and Assignments," for a complete description of each operator.

Constants

A constant is a number, a character, or a string of characters used as a value in a program. Since a constant's value does not change from execution to execution, constants are typically used to initialize program variables or to serve as fixed data, such as filenames and test values.

The C language has four kinds of constants: integer, floating point, character, and strings. The following sections define the format and use of each.

Integer Constants

An integer constant is a decimal, octal, or hexadecimal number that represents an integer value. A decimal constant has the form:

digits

where *digits* is one or more decimal digits (0 through 9). An octal number has the form:

0*odigits*

where *odigits* is one or more octal digits (0 through 7). The leading 0 is required. A hexadecimal number has the form:

0x*hdigits*

where *hdigits* is one or more hexadecimal digits (0 through 9 and either uppercase or lowercase a through f). The leading 0x is required. In all cases, whitespace characters must not be used between the digits of the constant.

The following examples illustrate the form of constants.

Decimal	Octal	Hexadecimal
10	012	0xa or 0xA
132	0204	0x84
32179	076663	0x7db3 or 0x7DB3

Integer constants always specify positive values. If negative values are required, the minus sign (-) can be placed in front of the constant to convert its value to a negative value. The minus sign is treated as an arithmetic operator.

Every integer constant is given a type based on its value. A constant's type determines what conversions must be performed when the constant is used in an expression or when the minus sign (-) is applied. Decimal constants always have signed types. A decimal constant in the range 0 to 127 has **char** type. In the range 128 to 32,767, the constant has **short int** type, and in the range 32,768 to 2,147,483,647, the constant has **long** type. Octal and hexadecimal constants always have unsigned types. Constants in the range 0 to 255 have **unsigned char** type. In the range 256 to 65,535, constants have **unsigned short int** type, and in the range 65,536 to 4,294,967,295, they have **unsigned long** type. Types are described in Chapter 3, "Declarations"; conversions in Chapter 4, "Expressions and Assignments".

You can direct the C compiler to force any integer constant to have **long** type (**signed** or **unsigned**, depending on the constant) by appending the letter l or L to the end of the constant as shown by the following examples.

Decimal	Octal	Hexadecimal
10L	012L	0xaL or 0xAL
79l	0117l	0x4fl or 0x4Fl

Floating-Point Constants

A floating-point constant is a decimal number that represents signed real number, that is, a number whose value includes an integer, a fraction, and an exponent. A floating-point constant has the form:

[-] *digits*[.*digits*][E [-]*digits*]

where - is the optional minus sign, *digits* is one or more decimal digits (0 through 9), and *E* is the exponent symbol (a lowercase e can also be used). The integer in a floating point constant has the same form as a decimal integer constant. If a fraction is given, it must be immediately preceded by a decimal point (.), and if an exponent is given, it must be immediately preceded by the exponent symbol. Whitespace characters must not be placed between the digits or characters of the constant. The following examples illustrate the various forms.

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

An alternate form of the floating-point constant allows a fraction without a corresponding integer. The following examples illustrate this form.

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

Floating point constants, like integer constants, are always given a type. All floating point constants receive **double** type unless the value is clearly being assigned to a variable with **float** type. In such cases, it is given **float** type.

Character Constants

A character constant is a letter, digit, or punctuation character enclosed in single quotation marks (' '). The value of a character constant is the character itself. Character constants consisting of more than one character or escape sequence are not allowed.

A character constant has the form:

'*c*'

where *c* can be any character from the C character set (including any escape sequence) except a single quotation mark ('), a backslash (\), or a newline character. If you wish to use a single quotation mark or backslash character as a character constant, you must precede it with a backslash as shown in the following examples.

Constant	Value
'a'	lowercase a
'?'	question mark
'\b'	backspace
'\x1B'	ASCII escape character
'\''	single quotation mark
'\\'	backslash

Character constants have **char** type and consequently are sign-extended in type conversions (see chapter 4, "Expressions and Assignments" for more on type conversions).

String Constant

A string constant is a sequence of letters, digits, and symbols enclosed in double quotation marks. A string constant has no single value. Instead, it is treated as an array of character constants with each element in the array equal to a single character value.

A string constant has the form:

"*Characters*"

where *characters* is one or more characters from the C character set except the double quotation mark ("), backslash (\), and newline character. If you wish to use the double quotation mark or backslash character within a string constant, you must precede it with a backslash as shown in the following examples.

```
"This is a string constant."
"Enter a number between 1 and 100\n Or press Enter"
"First\0Second"
"\"Yes, I do,\" she said."
```

A string constant can contain any number of characters. If the constant is longer than can fit on one line, you can break the string into two by using the backslash followed by the newline character. For example, the string constant

```
"Long strings can be broken\
into two pieces."
```

is identical to the string

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

Each string in a program is considered to be a distinct item, so even if there are two identical strings in a program they each receive distinct storage space.

String constants have the type **char []** . This means a string is an array whose elements have **char** type. The number of elements in the array is the number of characters in the string constant plus one, since the null character stored after the last character counts as an array element.

Identifiers

Identifiers are the names that you supply for the variables and functions used in a given program. You create an identifier by declaring it with the associated variable or function. You use the identifier in later statements within the program to refer to the given item. (Declarations are described in Chapters 3 and 6).

An identifier is a sequence of one or more letters, digits, or underscores (_) that begins with a letter or underscore. Any number of characters are allowed in a given identifier, but only the first 31 characters are used by the compiler. (Other programs that read the compiler output can use fewer characters.) Use leading underscores with care. Identifiers beginning with an underscore may conflict with the names of hidden system routines and produce errors.

The following examples illustrate the form of an identifier.

```
i
cnt
temp1
top_of_page
_skip12
```

Uppercase and lowercase letters are considered distinct characters and can be used to to make distinct identifiers that otherwise have the same spelling. For example, all of the following identifiers are unique.

```
add
ADD
Add
aDD
```

The C compiler does not allow identifiers that have the same spelling as a C language keyword. Keywords are described in the next section.

The linker may further restrict the number and type of characters for globally visible symbols.

Keywords

Keywords are predefined identifiers that have special meaning to the C compiler. Keywords indicate actions to be taken or attributes to be associated with given declared items. The following are keywords.

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

Two special identifiers **near** and **far** are reserved as keywords for use in some implementations of the compiler.

A keyword cannot be redefined and cannot be used in any other way than explicitly defined for it. However, you can specify text to be substituted for keywords using C preprocessor directives. See Chapter 7, "Preprocessor Directives."

Comments

A comment is a sequence of characters treated as a single whitespace character by the compiler but otherwise ignored. A comment has the form:

/* *characters* */

where *characters* can be any combination of characters including newline characters but excluding the combination */ .

Use comments to document the statements and actions of a C language source program. Comments can:

1. Appear anywhere a whitespace character is allowed.

2. Contain any combination of C keywords. These keywords are ignored by the compiler.

3. Occupy more than one line.

The following examples illustrate the form of comments.

```
/* Comments can be used to separate identifiers. */

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

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

Since all text within a comment is ignored, comments can be used to suppress compilation of portions of a program. However, comments cannot contain nested comments. For example, the comment:

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

causes an error since the comment stops at the the first */ . When using comments to suppress compilation, nested comments must be carefully removed. For suppression of a large portion of a program, use the **#if** preprocessor directive (see Chapter 7, "Preprocessor Directives").

C programmers generally rely on run-time libraries to perform a number of tasks such as input and output, screen manipulation and process control. The run-time library functions available for use in C programs are discussed in the *IBM Personal Computer XENIX Programmer's Guide to Library Functions.* The *CC* chapter of the *IBM Personal Computer XENIX Software*

Development Guide describes how to compile and link C source files. It also lists the command line options that are available. The *IBM Personal Computer XENIX Software Development Guide* also contains information specific to the implementation of C on the IBM Personal Computer XENIX system.

Tokens

When the compiler processes a program, it breaks the program down into groups of characters known as "tokens." A token is a unit of program text that has meaning to the compiler and that cannot be broken down further. The operators, constants, identifiers, and keywords described in this chapter are examples of tokens.

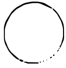
Tokens are delimited by whitespace characters and by other tokens, such as punctuation symbols. To prevent the compiler from breaking an item down into two or more tokens, whitespace characters are prohibited between the characters of identifiers, multicharacter operators, and keywords.

When the compiler interprets tokens, it incorporates as many characters as possible into a single token before moving on to the next token. Because of this behavior, tokens not separated by whitespace may not be interpreted in the way you expect. For example, in the following expression, the compiler first makes the longest possible operator (++) from the three plus signs, then processes the remaining plus sign as an addition operator (+).

```
i+++j
```

This expression is interpreted as (i++)+(j), not (i)+(++j). Use whitespace and parentheses to clarify your intent in such cases.

Chapter 2. Program Structure



Contents

Introduction 2-3

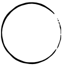
Source Program 2-3

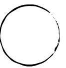
Source Files 2-5

Program Execution 2-7

Directives 2-9

Scope and Visibility 2-10





Introduction

This chapter describes the structure of C language source programs and defines some of important terms used later in this manual to describe the C language. In particular, it describes

- Source Programs
- Source Files
- Program Execution
- Directives
- Blocks and Visibility

This chapter describes features of the C language described in detail in other chapters. In particular, the syntax and meaning of directives are described in Chapter 7, "Preprocessor Directives". The syntax and meaning of declarations and definitions are described in Chapter 3, "Declarations", and Chapter 6, "Functions".

Source Program

Syntax

[*directives*]
[*declarations*]
[*definitions*]

Description

A C source program is a collection of one or more directives, declarations, and definitions. "Directives" are preprocessor directives. These direct the C preprocessor to perform specific actions on the text of the program, such as substitution of identifiers and inclusion of text from files. "Declarations" are

type, variable, or function declarations. These define the names and attributes of items used in a program. "Definitions" are special declarations that define the initial values for the declared variables or executable statements for the declared functions.

A source program can have any number of directives, declarations, and definitions. Each must have have the appropriate syntax as described in this manual. They can appear in any order in the program. However, the order in which items are declared or defined does affect how they can be used within the program (see "Visibility and Scope" in Chapter 3).

A nontrivial program always contains at least one function definition. This definition defines the action to be taken by the program. Directives and other declarations are optional.

The following example illustrates a simple C source program.

Example

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

int z;           /* Variable declarations */
int w;

extern int printf();        /* Function declaration */

main ()            /* Function definition for "main" */
{
   z = y + x;          /* Executable statements */
   w = y - x;
   printf("z= %d \n w= %d \n", z, w);
}
```

This source program defines four variables and two functions. The variables x and y are defined with variable definitions; z and w with declarations. Variable definitions assign initial values to the variables. The function "printf" is defined with a function declaration; "main" with a function definition. A declaration defines just the name and return type; a function definition gives the executable statements as well.

Source Files

A C source program can be divided into one or more source files. A C source file is a text file that contains all or part of a C source program. The C compiler reads C language source files and compiles the statements found in them. Each compiled source file is called an object file. The object files of a program can be linked and executed using a suitable linker/loader.

Divide large source programs into several different source files. This makes the program easier to develop, debug, and maintain. Using several source files also promotes the creation of a library of useful functions, declared in separate source files, that can be used by any number of programs.

Follow these rules when creating C language source files:

1. A source file need not contain a complete C source program. It can, for example, contain just a few of the functions needed by the program. In this case, all source files that make up the program must be compiled individually and then linked, or inserted into the source before compilation by using the **include** directive.

2. A source file need not contain a function declaration. It is often useful to place variable definitions in one source file and simply declare these variables in the source files that must use them. This makes the definitions easy to find and modify if necessary.

3. A source file must contain complete declarations. This means the declaration of a function or other large items, such as structures, must not be split between two files.

4. Directives in a source file apply to that source file only. If a common set of directives are to be applied to a source program, then all source files in the program must contain these directives.

5. A source file must contain declarations for all functions called from the file but defined in other source files. If the function returns an **int** type value the declaration is optional.

6. A source file must contain declarations for all variables used in the file but defined in other source files.

The following is an example of a C source program contained in two source files. The "main" and "max" functions are assumed to be in separate files. The main function is the program entry point, that is, program execution starts here.

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

extern int max();   /* External function declaration
                       max is assumed to be in
                       another source file. */

main ()             /* Function definition of main */
{
        int x=1,y=2,w=3,z;

        z = max(x,y);
        w = max(z,w);
}
```

In this source file, the function "max" is declared to be an externally defined function. This means its definition exists in another source file. The function definition for "main" includes function calls to "max".

```
/****************************************
        Source file 2 - max function
 ****************************************/
int max (a, b)      /* Function definition for max */
int a, b;
{
          if ( a>b )
                    return (a);
          else
                    return (b);
}
```

This source file contains the function definition for "max". This definition satisfies the external declaration given in the first source file. Once the source files are compiled, they can be linked and executed as a single program.

Program Execution

Every program has a main program function. This function serves as the starting point for program execution and usually controls execution of the program by directing the calls to other functions in the program. A program usually stops executing at the end of the main function, although it may stop at other points in the program, depending on the execution environment.

Traditionally, the main program function is named "main," and many operating systems require this name for the main function. However, the C language does not explicitly define the name of the main function and imposes no restrictions when naming it.

A source program usually has other functions if the main function needs to perform one or more specific tasks several times. The main function can call these functions and perform the task. Calling a function causes execution to begin at the first statement in that function. The function returns control when a **return** statement or the end of the function is encountered.

All functions, including the main function, can have parameters. Functions called by other functions receive values for the parameters from the calling functions. Parameters of the main

function may receive values passed to the main function from outside the program (for example, from the command line when the program is executed.)

Traditionally, the first 2 parameters of the main function are passed with the names argc and argv. The first parameter holds the total number of arguments passed to the function, and the second parameter is an array of pointers, each element of which points to a string representation of an argument passed to the main function. The operating system supplies values for the argc and argv parameters, and the user supplies the actual arguments to main. The argument-passing convention in use on a particular system is determined by the operating system rather than by the C language; see the *IBM Personal Computer XENIX Basic Operations Guide* for details.

Formal parameters to functions must be declared when the function is defined. Function definitions and declarations are described in detail in Chapters 3 and 6.

Directives

Directives in a source program perform special actions such as inserting lines of program text from other places into the source program or replacing names with specific values. Directives are not a part of the C language. They are instead instructions to the C language preprocessor that processes source programs before any compilation begins. This means the action of a directive takes place before compilation and never during execution of the program.

Directives are often used to expand or modify a source program. One common action is to insert a collection of function declarations from a standard place. Once inserted, these functions are considered part of the original source program and can be called from the **main** function or other explicitly defined functions. For example, the following program contains an **include** directive, which inserts declarations from the standard file stdio.h.

```
#include <stdio.h>

main ()
{
        printf("This program calls the printf\n");
        printf("function of a standard input\n");
        printf("and output library\n");
}
```

Directives can appear anywhere in the source program. The form and action of each directive is described in detail in Chapter 7.

Scope and Visibility

A "block" is a clearly defined collection of C language declarations and statements. A block can be a function body or a compound statement. A function body is the declarations and statements of a function definition. Compound statements are declarations and statements enclosed in braces (see Chapter 5, "Statements").

Blocks can be nested. Function bodies can contain nested compound statements, and compound statements can contain other compound statements. This creates "levels of nesting " in C programs. There are two levels of nesting: global and block.

The "global level" includes declarations and statements in a complete source file. All function definitions are at the global level as well as all variable definitions and declarations not made in any function. All blocks in the source file are nested within this level.

The "block level" includes the statements and declarations in a function body or compound statement. There can be several different block levels. All function bodies are at the first block level. All compound statements within function bodies are at the second level. Compound statements within compound statements are at the third, and so on.

Blocks and nesting affect how and when functions and variables can be used in a program. A variable or function is "visible" in a function if its type and name are known in the block. Visibility depends on where the declaration or definition appears in the source file and in what blocks it appears.

In general, a variable or function is visible from the point it is first declared or defined to the end of the block in which its declaration or definition appears. If it is declared at the global level, it is visible to the end of the source file. A variable or function is also visible in all nested blocks within their visibility. A variable or function is not visible in blocks that precede its first declaration or definition, or in nested blocks which contain declarations which redefine the variable's or function's name.

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

```
int i = 1;          /*  i declared at global level */
               /*  It is known to end of the source file */
main ()
{
        int j;            /* j declared in function body */
             /*  It is known to the end of the function */

        j = i;            /* i is visible in main */

        {
                int i;     /* i redeclared in nested block */
                    /*  global level i is no longer visible */

                i = j;      /* j is visible in this block */

        }                   /* nested i is not visible */
                            /* beyond end of the block */

}                 /* j is not visible beyond the end */
                  /* of the function                 */
```

Visibility does not extend outside a source file. However, there are ways to access variables and functions declared and defined in other source files by using implicit and explicit storage classes.

Visibility, storage classes, and scope (related to storage class) are all described in more detail in the section "Visibility and Scope" in Chapter 3.

Chapter 3. Declarations

Contents

Introduction **3-3**

Storage Class Specifiers **3-4**
- Automatic Class 3-4
- Register Class 3-5
- Static Class 3-5
- External Class 3-6

Type Specifiers **3-7**
- Fundamental Types 3-8
- Enumeration Types 3-9
- Structure Types 3-10
- Union Types 3-12

Declarators **3-13**
- Pointer Modifier 3-14
- Array Modifier 3-15
- Function Modifier 3-17
- Complex Declarators 3-17

Type Declarations **3-20**
- Enumeration Declarations 3-21
- Structure Declarations 3-22
- Union Declarations 3-24

Variable Declarations **3-26**
- Simple Variable Declarations 3-27
- Array Declarations 3-28
- Structure Declarations 3-30
- Union Declarations 3-31
- Pointer Declarations 3-33

Function Declarations **3-35**

Typedef Declarations **3-36**

Initialization **3-38**
Fundamental Types 3-38
Pointer Types 3-39
Aggregate Types 3-40
String initializers 3-43

Visibility and Scope **3-44**

Global and External Variables **3-45**

Static Variables **3-46**

Global and External Functions **3-47**

Static Functions **3-47**

Type names **3-48**

Introduction

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

[*sc-specifier*] [*type-specifier*]*declarator*
[*initializer*]
[,*declarator* [*initializer*]] . . .

where *sc-specifier* is a storage class specifier, *type-specifier* is the name of a defined type, *declarator* is an identifier optionally modified to declare a pointer, array or function, and *initializer* is a value or sequence of values to be assigned to the variable being declared.

This chapter describes each of the above declaration components. In addition, it discusses the placement of declarations within a program. The location of a declaration within the set of files that constitute a program can affect the declared variable's "scope" (the region of the program in which the variable is defined) and "visibility" (the sections of the program that can use the variable).

The topics of the chapter are as follows:

- Storage class specifiers
- Type specifiers
- Declarators
- Declarations
- Initializations
- Scope and visibility
- Type names

Although function declarations are presented in this chapter, function definitions are described in Chapter 6.

Storage Class Specifiers

A storage class specifier is a name for one of four possible storage classes. A storage class determines how and when a given variable is allocated storage as well as where in the program the variable name can be used. The storage classes are:

automatic
register
static
external

The following sections describe these storage classes in detail.

Automatic Class

Syntax

auto

Description

The **auto** storage class specifies that storage will exist for the duration of a function or compound statement. Variables with automatic storage class have storage while execution remains in the function or compound statement in which the variable is declared, but lose this storage when execution leaves.

The **auto** storage class can only be used in declarations in a function body or compound statement. It cannot be applied to global or function declarations. The storage class does not affect visibility (see the section "Visibility and Scope" later in this chapter).

The **auto** storage class is the default storage class for all variables declared within a function body or compound statement and for all formal parameters.

Register Class

Syntax

register

Description

The **register** storage class specifies that storage will exist for the duration of a function or compound statement and that this storage will be heavily used. Variables with register storage class sometimes have storage in CPU registers rather than in actual memory in order to reduce the access time for the given variable and to improve program execution time.

Storage allocation depends on available registers and on the variable's type. Storage allocation in registers is only guaranteed for **int** and pointer type variables. The number of available registers is machine dependent. Variables not allocated in registers have storage in memory.

The **register** storage class can only be used in declarations within a function body or compound statement. It cannot be applied to global or function declarations. It can be applied to formal parameter declarations. The storage class does not affect visibility (see the section "Visibility and Scope" later in this chapter).

Static Class

Syntax

static

Description

The **static** storage class specifies that storage will exist for the duration of the program. Variables with static storage class do not lose their storage when execution leaves the function or compound statement in which the variable is declared. Thus,

these variables guarantee access to the same storage for the lifetime of the program. Static storage is initialized to zero if no explicit initial value is given.

The **static** storage class does not affect the visibility of the variable unless it is declared at a global level. In this case, the visibility is restricted to the given source file (see the section "Visibility and Scope" later in this chapter).

The **static** storage class can be applied to a function declaration. In this case, the visibility of the function is restricted to the given source file. The storage class has no other affect.

The **static** storage class can be used in declarations at any level. It cannot be applied to formal parameter declarations.

External Class

Syntax

extern

Description

The **extern** storage class specifies that storage has been allocated elsewhere in the program and that this storage will exist for the duration of the program. Variables with external storage class require a corresponding global definition, defining the actual storage, somewhere in the program.

The **extern** storage class can be applied to a function declaration. In this case, the function is assumed to be explicitly defined (using the same name and return type) elsewhere in the program.

The **extern** can be used at any level in the program. It must not be used with formal parameter declarations. This storage class does not affect the visibility of the variable or function (see the section "Visibility and Scope" later in this chapter).

Type Specifiers

A type specifier is a name for a data type. The C language provides definitions for a number of basic data types, called the "fundamental" types. They are:

char
short int
long int
unsigned char
unsigned short int
unsigned long int
float
double
void

The **char, short int,** and **long int** types, together with their **unsigned** counterparts, are called "integral" types. **Float** and **double** are the "floating point" types. The **void** type applies only to functions that return no values.

In addition to the fundamental types, the user can declare "enumeration," "structure," and "union" types. An enumeration type defines a set of constant integer values and associates a name with each element of the set. Structure and union types define collections of variable values that can have different types. Structure and union types, together with arrays (discussed later in this chapter), are known as "aggregate" types.
You can declare additional type specifiers by using a **typedef** declaration (see the section "Typedef Declarations" later in this chapter).

Fundamental Types

The type specifier for each of the fundamental types consists of the type name or an abbreviation of the type name. The type specifiers and legal abbreviations for all fundamental types are summarized in the following table.

Type specifier	Abbreviation
char	—
short int	short
long int	long
unsigned char	—
unsigned short int	unsigned short
unsigned long int	unsigned long
float	—
long float	double
void	—

The type specifier **int** is a legal abbreviation, but it may not be portable, since **int** is equivalent either to a **short int** or a **long int,** depending on the implementation. Similarly, **unsigned** and **unsigned int** are equivalent either to the **unsigned short** or the **unsigned long** type specifiers. Wherever the type specifiers **int** or **unsigned** are used in defining features of the C language (for instance, in defining an **enum** type), the definition of **int** and **unsigned int** in a particular implementation determines the actual storage.

The next table summarizes the storage associated with each fundamental type and gives the range of values that can be stored in a variable of each type. Since the **void** type refers to a function returning no value, it has no associated storage or range.

Type	Storage	Range
char	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
long	4 bytes	-1,073,741,824to 1,073,741,823
unsigned char	1 byte	0 to 255
unsigned short	2 bytes	0 to 65,535
unsigned long	4 bytes	0 to 2,147,483,648
float	4 bytes	IEEE standard notation
double	8 bytes	IEEE standard notation
void	—	—

The **char** type stores a letter, digit, or symbol belonging to the C character set. The integer value of a character is the ASCII code corresponding to that character. Since a variable of **char** type is interpreted as a signed 1-byte integer, any value in the range -128 to 127 can be stored in a **char** variable, although only the values from 0 to 127 have character equivalents.

Enumeration Types

An enumeration type defines a set of named, constant **int** values (the enumeration set) and associates a name with each value of the set. A variable belonging to an enumeration type stores any one of the values defined by that enumeration type. Thus, the storage associated with a variable of an enumeration type is the storage required for a single **int** value. The name of an enumeration constant is equivalent to its value and can be used anywhere the value is required.

An **enum** type specifier has one of two forms:

enum [*tag*]{*enum-list*}
enum *tag*

The first form specifies the set of values for the enumeration type with *enum-list* and optionally names that enumeration type with *tag*.

An enumeration tag is simply an identifier used for naming an enumeration type. An *enum-list* has the form:

identifier [= *constant-expression*]

.
.
.

Each identifier of the *enum-list* names a value of the enumeration set; the optional "= *constant-expression*" clause specifies a constant integer value to be associated with that identifier. If the "= *constant-expression*" clause is omitted, a default value is associated with the identifier. Enumeration lists are discussed in more detail in the "Enumeration Declarations" section later in this chapter.

Once an enumeration type has been named, the second form of the enumeration type specifier can be used. In the second form, a tag referring to a declared enumeration type follows the **enum** keyword.

Structure Types

A structure type defines a sequence of variable values (called "members" of the structure) that can have different types. A variable belonging to a structure type holds the entire sequence defined by that structure type.

A structure type specifier has one of two forms:

struct [*tag*]{*struct-decl-list*}
struct *tag*

In the first form, the members of the structure are specified with the *struct-decl-list* and the structure type is optionally named with *tag*. A structure tag is simply an identifier used for naming a structure type. A *struct-decl-list* is a list of declarations. The declarations in the list can be variable declarations or bit field declarations.

Variable declarations in a structure declaration list have the form:

type-specifier declarator[, declarator . . .];

Storage class specifiers must not appear in a structure declaration list. Bit field declarations have one of two forms:

unsigned [*identifier*] : *constant-expression*;

Structure declaration lists, including bit field declarations, are discussed in more detail in the "Structure Declarations" section later in this chapter.

In the second form of the structure type specifier, a tag referring to a structure type follows the **struct** keyword.

A variable of structure type is allocated storage by giving each of its members appropriate storage for its type. Allocation of storage proceeds in order from the first member in the structure definition to the last. Thus, the first member has the lowest memory address and the last member the highest.

The storage for each member of a structure begins on a memory boundary appropriate to its type. Therefore, unnamed blanks can occur between the members of a structure in memory. Bit fields are not stored across **int** boundaries. Either a bit field is packed into the space remaining in the previous **int** or it begins a new **int.** An unnamed bit field with a length of 0 causes the member following it in the declaration list to be aligned at an **int** boundary.

Union Types

A union type defines a collection of variable values (called "members" of the union) that can have different types. A variable belonging to a union type holds at most one of its members at any given time.

A union type specifier has one of two forms:

union [*tag*]{*union-decl-list*}
union *tag*

In the first form, the members of the union are specified with the *union-decl-list* and the union type is named with *tag*. A union tag is simply an identifier used for naming a union type. A *union-decl-list* is a list of variable declarations. A variable declaration in a union declaration list has the form:

type-specifier declarator [,*declarator*] . . . ;

Storage class specifiers can not appear in the union declaration list. Union declaration lists are discussed in more detail in the section "Union Declarations" later in this chapter.

In the second form of the union type specifier, a tag referring to a union type follows the **union** keyword.

So that a variable of union type can accommodate any of its members, the amount of storage associated with a union type is the amount of storage required for the longest member of the union.

Declarators

A declarator specifies the name of the type, variable, or function being declared. It also modifies the given type-specifier, giving that type an array, pointer, or function attribute. Declarators have the following forms:

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

A declarator has an identifier and zero or more modifiers. When a declarator consists of an unmodified identifier, the item being declared has an unmodified type. If the identifier has a preceding asterisk (*), the type is modified to be a "pointer" type. If the identifier has following brackets ([ ]), the type is modified to an "array" type. If the identifier has following parentheses, the type is modified to a "function returning" type.

Any declarator can be parenthesized. Parentheses specify a particular interpretation of a "complex" declarator -- that is, a declarator containing more than one kind of modifier.

The following sections describe the pointer, array, and function modifiers. The last section describes the interpretation of complex declarators and the use of enclosing parentheses.

Pointer Modifier

An identifier is declared a pointer by prefixing the identifier with an asterisk:

**identifier*

A pointer type is declared to "point to" an object of a given type. For example, in the following declaration, the variable *p* is declared as a pointer to a **long :**

```
long *p;
```

A pointer can point to any type but **void.** Thus, a pointer can point to another pointer. For example:

```
long **q;
```

declares a pointer to a pointer to a **long.**

A variable declared as a pointer holds a memory address. The storage associated with variable of pointer type depends on the amount of storage required for a memory address on a given machine.

Array Modifier

An identifier declared as an array has one of the following forms:

identifier [ ]
identifier [constant-expression]

The constant expression in brackets, when it appears, specifies the size of the array. The constant expression can be omitted from an array declarator when:

1. The declarator is followed by an initialization.
2. When a formal argument to a function is being declared.
3. It is an external declaration of an array.

The elements of an array have a given type.

For example:

```
float x[3];
```

declares "x" to be an array of three **float** values.

The elements of an array can be of any type, except that arrays of functions are not allowed. An array can have elements of array type. For example:

```
float x[3][3][4];
```

declares "x" to be a three-dimensional array of **float** values, with a total of 36 elements. Each element of "x" is a 3 x 3 x 4 array; each element of x[3] is a 3 x 4 array; each element of x[3][3] is a 4-element array; and finally, each element of x[3][3][4] is a **float.**

When a multi-dimensional array is declared, the sizes of the second and succeeding dimensions must be specified in all cases.

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

Function Modifier

An identifier declared as a function has the following form:

identifier ( );

A function is declared to return a value of a given type. For example:

```
long f( );
```

declares a function returning a **long** value. A function cannot return an array or a function, though it can return a pointer to any type. Thus, a construction such as long f( )( ) is illegal.

Complex Declarators

Various combinations of the above modifiers can be applied to a single identifier. Some combinations are illegal, since an array cannot be composed of functions and a function cannot return an array or a function.

In interpreting complex declarators, the array modifier [ ] and the function modifier ( ) have higher priority than * , and group left to right. For example:

```
int *var1[5];
```

declares *var1* to be an array of five pointers to **int** values.

Parentheses can be used to group modifiers in a way that forces a particular interpretation. For example, enclosing *var1 in parentheses alters the meaning of the above declaration:

```
int (*var1)[5];
```

Now *var1* is declared to be a pointer to an array of **int** values.

Similarly:

```
long *var2();
```

declares *var2* to be a function returning a pointer to a **long,** while:

```
long (*var2)();
```

declares *var2* to be a pointer to a function returning a **long.**

Although the elements of an array cannot be functions, they can be pointers to functions. Expanding the above example:

```
long ( *var2[ ] )( );
```

declares an array of pointers to functions returning **long** values.

A function cannot return an array, but it can return a pointer to an array. Thus:

```
float  (*var3( ) )[ ];
```

declares *var3* a function returning a pointer to an array of **float** values.

A pointer can point to another pointer, and an array can contain array elements. Thus:

```
int **var4[5][5];
```

declares *var4* to be an array of five elements; each element is a five-element array of pointers to pointers to **int** values. Parentheses can override this interpretation in various ways. For example:

```
int (**var4)[5][5];
```

declares *var4* to be a pointer to a 5 x 5 array of **int** values, while:

```
int (**var4[5])[5];
```

declares a five-element array of pointers to pointers to five-element arrays of **int** values. Finally:

```
int *(*var4[5])[5];
```

declares a five-element array of pointers to five-element arrays of pointers to **int** values.

Type Declarations

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

Enumeration, structure, and union types can also be declared while declaring variables or functions. See the section "Variable Declarations" for examples.
The following sections describe enumeration, structure, and union declarations in detail.

Enumeration Declarations

An enumeration declaration has the form:

enum [*tag*]{*enum-list*} ;

The optional enumeration *tag* names the type being declared.

An *enum-list* has the following form:

identifier [=*constant-expression*]
[, *identifier* [= *constant-expression*]

.
.
.

Each *identifier* names a value of the enumeration set. If no "= *constant-expression*" phrase appears in the enumeration declaration, the first identifier names the value 0, the next identifier names the value 1, and so on through the last identifier appearing in the declaration.

The = *constant-expression* phrase overrides the usual sequence of values. An identifier followed by = *constant-expression* is associated with *constant-expression*, which must be an integer constant. After one = *constant-expression* clause appears in the declaration, the next identifier in the declaration names *constant-expression* + *1*, unless it is explicitly given another value with = *constant-expression*.

Each identifier in an enumeration list must be unique within that list. It is not necessary for the identifiers in a list to be distinct from ordinary variable names or from identifiers in other enumeration lists.

The following is an example of an enumeration type declaration:

```
enum weather {
    sunshine,
    rain = 10,
    snow = 20,
    sleet,
    hail
} ;
```

This enumeration type is named "weather". The value 0 is associated with "sunshine" by default. The identifiers "rain" and "snow" are given the values 10 and 20, respectively. "Sleet" and "hail" have the values 21 and 22 by default. Each enumeration identifier can be used anywhere a constant integer value is expected.

Structure Declarations

A structure type declaration has the form:

struct [*tag*]{*struct-decl-list*};

The optional structure *tag* names the structure type being declared.

The *struct-decl-list* is a list of one or more declarations that specifies the members of the structure. A structure member may be declared as a variable. Variable declarations within structure declaration lists have the same form as the variable declarations discussed later in this chapter, with three restrictions. The storage class can not be specified for variables declared within a structure declaration list; the type must be specified; and a variable cannot be declared to have the type of the structure in which it appears (although it can be a pointer to that structure type).

A structure member can be declared as a bit field. The form of a bit field declaration is:

unsigned [*identifier*] : *constant-expression*;

A bit field consists of the number of bits specified by *constant-expression.* The constant expression must be a non-negative integer value. Its maximum value is the number of bits in an **int** for a given implementation. The optional *identifier* names the bit field. Every bit field, named or unnamed, must be declared **unsigned.** An unnamed bit field whose width is specified as 0 has a special function: it guarantees that storage for the member following it in the declaration list begins on an **int** boundary.

Each identifier in a structure declaration list must be unique within that list. It is not necessary for the identifiers in the list to be distinct from ordinary variable names or from identifiers in other structure declaration lists.

The following is an example of a structure type declaration:

```
struct sample {
    int i, j, k;
    char c;
    float *pf;
    long la[10];
    struct sample *next;
} ;
```

The structure type is named *sample.* The members of the structure are, in order, the **int** values "i", "j", and "k"; a **char** value; a pointer to a **float** value; a 10-element array of **long** values; and a pointer to the structure type *sample.*

Bit field members of a structure are useful for storing single bit values such as flags.

For example, the structure type declared above can be modified to include bit field members:

```
struct newsample {
    int i, j, k;
    char c;
    unsigned flag1 : 1;
    unsigned flag2 : 1;
    unsigned flag3 : 1;
    unsigned flag4 : 1;
    float *pf;
    long la[10];
} ;
```

Union Declarations

A union type declaration has the form:

union [*tag*]{*union-decl-list*};

The optional union *tag* names the union type being declared. The *union-decl-list* is a list of one or more declarations specifying the members of the union. The declarations of a union declaration list are variable declarations. Variable declarations within union declaration lists have the same form as the variable declarations discussed later in this chapter, with three restrictions. The storage class cannot be specified for variables declared within a union declaration list; the type must be specified; and a variable cannot be declared to have the type of the union in which it appears (although it can be a pointer to that union type).

Each identifier in a union declaration list must be unique within that list. It is not necessary for the identifiers in a list to be distinct from ordinary variable names or from identifiers in other union declaration list. The following is an example of a union declaration:

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

The name of the union type is *jack*. The members of the union are, in order, a pointer to a **char** value, a **char** value, and an array of **float** values. The storage associated with the union type *jack* is the storage required for the 20-element array *f*, because *f* is the longest member of the union.

Variable Declarations

The following sections describe the form and meaning of declarations for variables. In particular, it describes declarations for:

Simple variables	Single value variables with integral or floating point type.
Arrays	Variables composed of a collection of elements that all have the same type.
Structures	Variables composed of a collection of members that may have different types.
Unions	Variables composed of several members of different types occupying the same storage space.
Pointers	Variables that point to other variables, that is, variables that contain locations (in the form of addresses) instead of values.

The general form for all variable declarations is given in the first section of this chapter. More than one variable can be defined in a given declaration by giving more than one declarator. Each declarator defines a unique variable that receives the same storage class and type as the other variables defined in the declaration but that has a unique name and attributes as defined by the declarator. Examples of multiple declarations are given in each of the following sections.

Storage classes in variable declarations are described later in this chapter.

Simple Variable Declarations

Syntax

type-specifier identifier ;

Description

A declaration for a simple variable defines the variable's name and type. The *identifier* is the variable's name and *type-specifier* gives the type. The *type-specifier* must be an integral or floating point type; no other types are allowed for simple variables.

Several variables can be defined in the same declaration by giving a list of identifiers separated by commas (,). In such cases, one variable is defined for each identifier in the list. Variables in this list will have the same type.

Examples

1. int x;

   Defines a simple variable *x*. This variable can be assigned any value in the set defined by **int** type.

2. long reply, flag;

   Defines two variables, *reply* and *flag*. Both variables have **long** type and can be assigned any large integer value.

3. enum {first, second, third} order;

   Defines a variable *order* that has an enumeration type. This variable may be assigned the values *first*, *second*, or *third* defined in the enumerated list.

Array Declarations

Syntax

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

Description

A declaration for an array defines the name of the array, the number of elements in the array, and the type of each element. In this declaration, the *identifier* is the array's name, *constant-expression* defines the number of elements, and *type-specifier* is the type of each element. The constant expression must be enclosed in brackets ([ ]). The *type-specifier* can be an integral or floating point type.

Use the second form of array declaration only:
1. If the array is initialized (see the section "Initializations" later in this chapter.)
2. If this is an external declaration of an array explicitly defined in another source file of the same program.
3. It is the formal argument to a function.

Arrays of arrays (or multiple dimensioned arrays) can be defined by giving a list of bracketed *constant-expressions.* The list has the form:

[*constant-expression*] [*constant-expression*] . . .

where each *constant-expression* defines the number of elements in a given dimension. Two-dimensional arrays have two bracketed expressions, three dimensional arrays have three, and so on.

Arrays of structures and arrays of unions can be defined by combining the structure or union type with bracketed constant expressions. Multi-dimensional arrays of structures and unions can be defined by giving more than one set of brackets.

Arrays of pointers can be defined by combining one or more bracketed constant expressions with the pointer modifier. Multi-dimensional arrays of pointers can be defined by giving more than one bracketed expression.

Note that array elements are stored in contiguous memory locations. The first element is stored at the lowest memory location, the last at the highest.

Examples

1. int scores[10];

   Defines a 10-element array named *scores*. Each element has int type.

2. char prompt[20], reply[25];

   Defines two arrays, *prompt* and *reply*. In this case, *prompt* has 20 elements and *reply* has 25. The elements of both arrays have char type.

3. float matrix[10][15];

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

4. struct { int x,y; } complex[100];

   Defines an array of structures. This array has 100 elements. Each element is a structure containing two members.

5. union { char x; int y; long z; } slice[20];

   Defines an array of unions. This array has 20 elements. Each element is a union containing three members of different types.

6. char *name[20];

   Defines an array of pointers. The array has 20 elements. Each element is a pointer to a **char** value.

Structure Declarations

Syntax

struct { *member-declaration-list* } *identifier* ;
struct *tag* { *member-declaration-list* } *identifier* ;
struct *tag identifier* ;

Description

A declaration for a structure defines the name of the structure, the number of members, and the names and types of the members. The *identifier* is the structure's name and the *member-declaration-list* defines the members of the structure. If *tag* is given but no *member-declaration-list* , the *tag* implies no members unless it has been explicitly declared. The exact syntax for member declarations and the meaning of the *tag* are described in detail in the section "Structure Declarations" in this chapter.

Structure members are stored in the same order as they are declared. The alignment and packing of structure members is machine dependent.

Examples:

1.
```
struct { int x,y; } complex;
```

Defines a structure named *complex*. This structure has two members, x and y. Both members have **int** and can be assigned integer values.

2.
```
struct record {
        char name[20];
        int id;
        long class;
} employee;
```

Defines a structure named *employee*. The structure has three members: *name*, *id*, and *class*--where *name* is a 20 element array and *id* and *class* are simple members with **int** and **long** type, respectively. The identifier *record* is the

structure tag. It can be used in another structure declaration to refer to the same list of members as given in this declaration.

3. struct record student, faculty, staff;

Defines three structures: *student*, *faculty*, and *staff*.
Each structure has the same list of members. The members are defined by the structure tag *record* that was defined in the previous example. Thus, each structure has three members: name, id, and class.

4.
```
struct {
 char icon;
 unsigned color : 4;
} screen[25][80];
```

Defines a two-dimensional array of structures named *screen*. The array contains 2000 elements, each element an individual structure. Each structure in this array has two members,*icon* and *color*. The icon is a simple **char** type member, but color is a bit field containing a 4-bit unsigned integer value. The bit field saves storage space when the array is allocated.

Union Declarations

Syntax

union { *member-declaration-list*} *identifier* ;
union *tag* { *member-declaration-list*} *identifier* ;
union *tag identifier* ;

Description

A declaration for a union defines the name of the union, the number of members, and the names and types of the members. The *identifier* is the union's name and the *member-declaration-list* defines the members of the union. If *tag* is given but no *member-declaration-list* , the *tag* defines the members of the union

instead. The exact syntax for member declarations and the meaning of the *tag* are described in detail in the section "Union Declarations" in this chapter.

All members of a union are stored in the same memory space and start at the same address. The compiler allocates enough memory space to store the largest member. This means there can be some unused memory space in the union when smaller members are being used. Since memory space is shared, only one member at a time can have a value. Assigning a value to a new member destroys the value of the previously assigned member.

Examples:

1.

```
union {
   int S;
   unsigned U;
} number;
```

Defines a union named *number* that has two members: S, a signed integer, and U, an unsigned integer. These members allow the current value of *number* to be stored as either a signed or an unsigned value.

2.

```
union names {
   char first[15], middle[15], last[20];
} employee;
```

Defines a union named *employee.* The structure has three members, *first*, *middle*, and *last*. Each member is an array of **char** type. The arrays can receive the same type of values, but not the same number of elements: *last* has 20 elements; *first* and *middle* have only 15. The identifier *names* is the union tag. It can be used in another union declaration to refer to the same list of members as given in this declaration.

3.

```
union names student, faculty, staff;
```

Defines three unions: *student*, *faculty*, and *staff*. Each union has the same list of members. The members are defined by the union tag *names* that is assumed to be defined in another union declaration.

4.

```
union {
   struct {
     char icon;
     unsigned color : 4;
   } window1, window2, window3, window4;
} screen[25][80];
```

Defines a two-dimensional array of unions named *screen*. The array contains 2000 elements, each element an individual union. Each union in this array has four members, *window1*, *window2*, *window3*, and *window4*, where each member is a structure. In this array, the entire screen is a composite of up to four different windows. Each element in the array has only one of the possible four values at any given time.

Pointer Declarations

Syntax

*type-specifier *identifier* ;

Description

A pointer declaration defines the name of a pointer variable and associates a type with the object to which the variable points. The *identifier* defines the variable's name, and the *type-specifier* gives the type of the object. The type can be any integral, floating point, structure, or union type.

A pointer to an array can be defined by enclosing the asterisk (*) and *identifier* in parentheses before giving the bracketed constant expression. In this case, the identifier has the form:

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

The same rules for *constant-expression* given above apply here.

Pointer declarations can also be used within the member declaration list of a structure or union. Pointers can be used in this way to create linked lists.

Pointers can be declared using forward structure references. A forward structure reference is the use of a structure tag before it has been declared. Such declarations define a pointer to a structure whose members are declared later in the program. Forward structure references let structures contain pointers to themselves. See the fourth example given below.

Pointers contain addresses rather than values. The amount of storage required for an address and the meaning of the address depends on the given implementation of the compiler.

Examples:

1. char *message;

Defines a pointer variable named *message*. It points to a variable with **char** type.

2. int *pointers[10];

Defines an array of pointers named *pointers*.
The array has 10 elements, each a pointer to a variable with **int** type.

3. int (*pointer)[10];

Defines a pointer variable named *pointer*.
It points to a ten element array. Each element in this array has **int** type.

4. structure linked {
       int token;
       struct linked *next, *previous;
   } list;

Defines a structure named *list* that contains two pointers and an integer. The pointers, named *next* and *previous*, point to structures that have the members defined by *linked* that is the same form as the structure *list*.

Function Declarations

A function declaration defines the name, return type, and storage class of a given function. Function declarations, also called forward declarations, do not define the function body or parameters. Instead they permit the function name and return type to be known before the function is actually defined. (Function definitions are described in detail in Chapter 6.)

Function declarations have the form:

[static | extern] *type-specifier identifier*( ) ;

The *identifier* is the function's name; the *type-specifier* gives the function's return type. It can be any type. If no type is given, **int** is assumed. The function's storage class can be **static** or **extern.** If no storage class is given, **extern** is assumed, until an explicit definition is given.

The return type can be modified by applying modifiers to the function identifier. A function can return pointers, but it cannot return arrays. Parentheses can be applied to the modifiers to make complex return types.

Examples:

1. int add();

   This example defines a function *add* that returns an **int** type value. The storage class is assumed to be **extern.**

2. char *strfind();

   This example defines a function *strfind* that returns a pointer to a **char** value.

3. static char test();

   This example defines a function named *test* that returns a **char** type value. This function has **static** storage class and is known only within the source file in which it is defined.

4. extern a(), b(), c();

   This example defines three functions *a*, *b*, and *c*. All three functions have **extern** storage class. Since no return type is explicitly given, **int** type is assumed.

Typedef Declarations

The form of a **typedef** declaration is:

typedef[*type-specifier*]*declarator* [,*declarator* . . .];

A **typedef** declaration is analogous to a variable or function declaration except that the **typedef** keyword appears in place of a storage class specifier. The declaration is interpreted exactly as are variable and function declarations, except that the identifier, instead of taking on the type specified by the declaration, becomes a synonym for the type. For example:

```
typedef int whole;
```

declares *whole* to be a synonym for **int**. Thus:

```
whole i;
```

is equivalent to:

```
int i;
```

Typedef does not create types; it creates synonyms for existing types or names for types that could be specified in other ways. Thus, if *j* is declared as a **whole** , and *k* is declared as an **int**, *j* and *k* are considered to have the same type. As a further example:

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

declares *group* to be a structure with three members. Since a structure tag, *club,* was also specified, either the **typedef** name or the structure tag can be used in declarations. The following declarations are equivalent:

```
group x, *g;
struct club x, *g;
```

The variable *x* is declared to be a structure of the type declared above, while *g* is a pointer to such a structure. Any type can be declared using typedef, including pointer, function and array types. For example, the declarations:

```
typedef void graphf();
typedef int (*pa)[];
```

provide the name *graphf* for a function returning no value and *pa* for a pointer to an array of **int** values.

```
graphf box;
pa *P;
```

Here *box* is declared as a function returning no value; *P* is declared as a pointer to a pointer to an array of **int** values.

Initialization

A variable is initialized (that is, set to an initial value) whenever an initializer is applied to the variable's declarator in the declaration. An initializer assigns the given initial value or values to the variable being declared.

Initialization has the form:

declarator = initializer

Only variables can be initialized; functions cannot. Variables of any type except **union** or **void** may be initialized. Variables with **extern** or **static** storage class can be initialized if desired. If not initialized, these variables are automatically initialized to 0.

Automatic and register variables can be initialized with constant or variable values. If the initialization is omitted for an automatic or register variable, the initial value of the variable is undefined. Thus, automatic and register variables must either be initialized or assigned to before they are used.

Initializers for fundamental, pointer, and aggregate types are described in the following sections.

Fundamental Types

An initializer for a variable belonging to a fundamental types has the form:

= *expression*

The value of *expression* is assigned to the variable. The conversion rules for assignment apply. For example, the following declaration initializes the variable *x*.

```
int x = 10;
```

Pointer Types

A pointer variable is initialized in the same manner as a variable of fundamental type. For instance:

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

initializes the pointer variable *px* to 0.

Aggregate Types

In its simplest form, an initializer for an aggregate type has the form:

= {*expression* [,*expression*] . . . }

The values are assigned in order to the members of the aggregate. The conversion rules for assignment apply. Only static variables or variables declared outside functions may be initialized; initializations of automatic and register aggregates are prohibited. Union variables cannot be initialized. The initializers for aggregate types must be constant expressions.

For example:

```
int P[ ]= { 2, 4, 6, 8, 10};
```

declares P as 5-element array and assigns the value 2 to the first element of P, the value 4 to the second element, and so on through the fifth element. When the size of the array is unspecified, the size defaults to the number of values in the initializer list. Since there are five initializer values, P is defined to be a five-element array.

A brace-enclosed initializer list can appear within another initializer list. This is useful for initializing aggregate members of an aggregate. For example:

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

declares P as a 4 x 3 array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row.

If there is no embedded initializer list for an aggregate member, values are simply assigned in order to each member of the subaggregate.

Thus, the above initialization is equivalent to:

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

Here the values are simply assigned in increasing subscript order to P. Since arrays are stored by row, initialization proceeds by row.

Braces can appear around any initializer in the list. For instance:

```
float x[3] = {
    {1},
    {2},
    {3}
};
```

has the same effect as:

```
float x[3] = {
            1, 2, 3
};
```

With fewer values in the initializer than there are members of the aggregate type, the remaining members are initialized to 0. For example, if the structure type *list* has been defined as:

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

then a variable *y* can be declared and initialized as follows:

```
struct list y = {
        1,
        2,
        3,
        {4, 4, 4}
};
```

The three **int** members of *y* are initialized to 1, 2, and 3, respectively; the three elements in the the first row of *x* are initialized to 4; and the elements of the remaining row of *x* are initialized to 0 by default.

In the following example the first column of *y* is initialized:

```
struct list y = {
        1,
        2,
        3,
        {4},
        {5}
        };
```

The three **int** members are initialized as above; then the initializer list {4} is used to initialize the first row of *x*. Since only one value appears in the list, the element in the first column is initialized to 4 and the remaining two elements in the row are initialized to 0, by default. Similarly, the element in the first column of the second row of *x* is initialized to 5, while the remaining two elements in the row are initialized to 0.

More initializers than members in the aggregate type causes an error.

String initializers

An array can be initialized with a string constant. For example:

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

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

If the array size is specified and the string is longer than the specified size of the array, the extra characters are simply discarded. The following declaration initializes S as a three-element character array:

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

In this case only the first three characters of the initializer are assigned to S. The character 'd' and the null character are discarded.

If the string is shorter than the specified size of the array, the remaining elements of the array are initialized to 0.

Visibility and Scope

Visibility and scope are special attributes that apply to the declared items of a C program. Visibility is the region within a program in which an identifier is known and can be used. Scope is the way in which storage is allocated for a variable during program execution. All declared items are subject to the rules of visibility and scope.

The following sections describe how the rules of visibility and scope affect variables and functions and how to modify or override these rules to achieve special results.

Global and External Variables

Variable declarations and definitions at a source file's global level have special properties that affect how the variables can be accessed. In particular, these declarations and definitions create one or three types of variable: global, or external.

A "global variable" is defined with a variable definition at the global level. Global variables have permanent storage for the duration of the program and have initial values. Global variables are special in that their storage is available for linking with external variables.

An "external variable" is any variable declared with the **extern** storage class at any level in a program. External variables have no storage of their own. Instead, they are linked to the storage of a global variable having the same name.

Global and external variables provide a way to access variables not visible in a given block or source file. An external variable is always linked to a global variable having the same name no matter where the external declaration occurs and no matter what source file contains the global variable definition.

A global variable must not be defined more than once in any given set of source files. Any number of external declarations can be given, but only one global definition is allowed for that variable.

The following program example illustrates how an external variable can provide access to a global variable that is not visible in a given part of a program.

```
int i = 1;      /* "i" is a global variable  */

main ()
{
     printf("%d\n",i);          /* prints 1 */
     {
            int i = 2;      /* "i" is redefined;  the global */
                            /* "i" is no longer visible      */

            printf("%d\n",i);      /* prints 2 */
            {
                 extern int i; /* external "i" linked */
                               /* to global "i"       */

                 printf("%d\n",i);    /* prints 1 */
            }      /* external "i" no longer visible */

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

     }       /* local "i" no longer visible */

     printf("%d\n",i);         /* prints 1 */
}
```

Static Variables

A "static variable" is any variable declared with the **static** storage class. Static variables are similar to global variables in that they have permanent storage and an initial value, but static variables cannot be linked to external variables. Static variables are accessible only within the region of a program in which they are visible.

Global and External Functions

A "global function" is any function that has an explicit function definition in a source file of the program. An "external function" is any function declaration that either has the **extern** storage class, or has a corresponding global function definition in the same source file or in another file. External functions are like external variables in that they are linked to global functions having the same name. This linking allows use of the global function in one source file even though its definition is in another, or use in blocks in which it is not visible.

Static Functions

A "static function" is any function that has been defined with the **static** storage class. Static functions are like static variables in that they cannot be linked to external functions. Static functions can only be used in the blocks in which they are visible.

Type names

A "type name" specifies a particular data type. Type names are used in two contexts: in type casts and in **sizeof** operations (see Chapter 4).

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

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

type-specifier abstract-declarator

An *abstract declarator* is a declarator without an identifier. (Declarators are discussed earlier in this chapter.) Thus, an abstract declarator consists of pointer, array and/or function modifiers. Since the pointer modifier (*) always appears before the identifier in a declarator, while array ([ ]) and function (()) modifiers appear after the identifier, it is possible to determine where the identifier would appear in an abstract declarator and to interpret the declarator accordingly.

For example:

```
long *
```

is the type name for a pointer to a **long**, while:

```
float * ()
```

is the type name for a function returning a pointer to a **float.**

An abstract declarator can be complex, and parentheses can be used within the declarator. For example:

```
int (*)[5]
```

names a pointer to an array of five **int** values, while:

```
int (*)()
```

names a pointer to a function returning an **int.**

The abstract declarator "( )" is not allowed because it is ambiguous. The implied identifier could appear within the parentheses or before them, with a different result in each case.

Chapter 4. Expressions and Assignments

Contents

Introduction **4-5**

Operands **4-6**

Constants **4-6**

Identifiers **4-6**
- Integral and floating point identifiers 4-6
- Enumeration identifiers 4-6
- Structure and union identifiers 4-7
- Pointer identifiers 4-7
- Array identifiers 4-7
- Function identifiers 4-7

Strings **4-8**

Function calls **4-9**

Subscript expressions **4-10**

Member Selection Expressions **4-13**

Expressions **4-14**
- Expressions with Operators 4-15
- Type Cast Expressions 4-15
- Expressions in Parentheses 4-16
- Constant Expressions 4-16

Type Conversions **4-17**

Assignmnent Conversions **4-18**
- Conversions from Signed Integer Types 4-18
- Conversions from Unsigned Integer Types 4-20
- Conversions from Floating Point Types 4-22

Conversions from Enumeration Type 4-24
Conversions from Structure and Union Types 4-24
Conversions from pointer types 4-24
Conversions from Void Type 4-24

Type Cast Conversions 4-25

Operator Conversions 4-25

Function Call Conversions 4-26

Operators 4-27

Complement Operators 4-28
Arithmetic negation (-) 4-28
Bitwise complement (~) 4-28
Logical not (!) 4-28

Indirection and address of operators 4-29
Indirection (*) 4-29
Address of (&) 4-29

The sizeof Operator 4-30

Multiplicative Operators 4-31
Multiplication (*) 4-31
Division (/) 4-31
Remainder (%) 4-31

Additive Operators 4-32
Addition (+) 4-32
Subtraction (-) 4-33
Pointer and integer combinations 4-33
Overflow 4-35

Shift operators 4-35

Relational operators 4-36

Bitwise Operators 4-37
Bitwise AND (&) 4-37
Bitwise Inclusive OR (|) 4-37
Bitwise Exclusive OR (^) 4-37

Logical Operators **4-38**
Logical AND (&&) 4-38
Logical OR (| |) 4-39

Sequential Evaluation Operator (,) **4-40**

The Conditional Operator (?:) **4-41**

Assignment Operators **4-42**

Unary Increment and Decrement **4-44**

Simple Assignment **4-45**

Compound Assignment **4-46**

Precedence **4-47**

Side Effects **4-51**

Introduction

This chapter describes how to form expressions and make assignments in the C language. The topics covered in the chapter are:

- Operands
- Type conversions
- Operators
- Assignments
- Precedence
- Side effects

An expression is a combination of operands and operators that yields (or expresses) a single value. An operand is a constant or variable value that is manipulated in the expression. Each operand of an expression is also an expression, since it represents a single value. An operator specifies how the operand or operands of the expression are manipulated.

The type of an operand can be converted to a different type in certain contexts. Type conversions can take place in assignments, type casts, function calls, and operations.

In C, assignments are considered expressions. An assignment yields a value: its value is the value being assigned. In addition to the simple assignment operator (=), C offers a number of more complex assignment operators that both transform and assign their operands.

The "precedence" of operators affects the grouping of operands in an expression. "Side effects" are changes that take place as a result of an expression's evaluation.

Operands

A C operand can be a constant, an identifier, a string, a function call, a subscript expression, or a member selection expression. An operand can also be an expression. The type of value yielded by each kind of operand is discussed below.

Constants

An operand that is a constant has the value and type of the constant value. A character constant has **int** type. An integer constant is signed; it can be either **int** or **long,** depending on its size and how the value was specified. A floating constant is a **double.**

Identifiers

An operand can be an identifier. An identifier is a name for a variable, constant, or function. Every identifier has an associated type; the value of an identifier depends upon its type, as follows.

Integral and floating point identifiers

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

Enumeration identifiers

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

Structure and union identifiers

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

Pointer identifiers

An identifier declared as a pointer represents a pointer to the specified type.

Array identifiers

An identifier declared as an array represents a pointer whose value is the address of the first element of the array. The type addressed by the pointer value is the type of the first element of the array. The address of an array's first element is a constant; thus, the pointer value represented by an array identifier is a constant.

Function identifiers

An identifier declared as a function represents a pointer whose value is the address of the function. The type addressed by the pointer is a function returning a value of some specified type. The address of a function is a constant; thus, the value represented by a function identifier is a constant.

Strings

An operand can consist of a string. A string is a list of characters enclosed in double quotes:

"*string*"

A string expression represents a pointer whose value is the address of the first character of the string. The type addressed by the pointer is **char.** Since the address of the first character of the string is a constant, the value represented by a string expression is a constant.

The last character of a string is always the null character, \0 . The null character is not visible in the string expression, but it is present as the last position of the string in memory. Thus, the string "abc" actually has 4 characters rather than 3.

Function calls

A function call has the form:

identifier (*expression-list*)

where *identifier* is the name of the function to be called, and *expression-list* is a list of expressions whose values, the actual arguments, are passed to the function. The expression list can be empty. The number of expressions in the list depends on the number of formal arguments specified in the function definition. An error results if the number of expressions in an expression list does not match the number of formal arguments in the function.

A function call expression has the value and type of the function's return value. If the function's return type is **void,** the function call expression also has **void** type. If control returns from the called function without execution of a **return** statement, the value of the function call is undefined.

When encountered in a program, the function call evaluates the expressions in the expression list, passes these values to the function, then passes execution control to the first statement in the function. The expressions in the expression list can be evaluated in any order. However, the first expression in the list always corresponds to the first formal argument of the function; the second expression corresponds to the second formal argument, and so on through the end of the list.

Certain type conversions are routinely performed in function calls; see Function Call Conversions later in this chapter. However, conversions do not take place between the actual arguments to a function and the formal arguments of the function. Type mismatches between actual and formal arguments can produce unexpected results.

Subscript expressions

A subscript expression has the form:

operand[*expression*]

The *operand* may be any operand representing a pointer value. The *expression* in brackets represents an integer value. Subscript expressions refer to an element of an array; thus, the pointer value is usually a pointer to an array.

To evaluate a subscript expression, the integer value is added to the pointer value. According to the conversion rules of the addition operator (discussed later in this chapter) the integer value is converted to an address offset by multiplying it times the length of the type addressed by the pointer. After conversion, an integer value *i* expresses *i* memory positions of the length specified by the pointer type. When added to the pointer value, the result is a new pointer value expressing the address *i* positions from the original address. The type addressed by the resulting pointer value is the type addressed by the original pointer value.

If the resulting pointer value addresses an array, the result of the subscript expression is that pointer value. Otherwise, the indirection (*) operator (discussed later in this chapter) is applied to the pointer value to yield the value residing at that address.

Notice that a subscript expression such as:

A[0]

yields the value of the first element of A, since the offset from the address represented by A is zero. Similarly, an expression such as:

A[5]

refers to the element offset five positions from the address represented by A or the *sixth* element of the array.

A subscript expression can be subscripted:

operand[*expression*][*expression*] . . .

Subscript expressions group left to right; the first subscript expression is evaluated first, and the result is used as the operand for the next subscript.

Expressions with multiple subscripts are used to refer to elements of multi-dimensional arrays. A multi-dimensional array is an array whose elements are arrays. If A is a 3-dimensional array, the first element of A is an array with 2 dimensions. The integer value *i* is multiplied times the length of that 2-dimensional array element to produce an address offset. This converted value is added to the pointer value A to yield a pointer to the *i*th element of A, which is a 2-dimensional array.

For example, if the array A is declared with 3 dimensions as follows:

```
int A[3][4][6];
```

the first round in the evaluation of a subscript expression such as:

```
A[2][1][0];
```

is the evaluation of:

```
A[2]
```

The value of A[2] is the address of the third element of A. The type of element addressed is 4 x 6 integer array.

The address yielded by the first subscript expression forms the operand for the second subscript expression. The integer value "1" is converted by multiplying it times the length of the 4 x 6 array element. The converted value is added to the pointer value yielded by A[2]; the result is the address of the second element of the 4 x 6 array. The type addressed by the pointer result is a 6-element array.

Finally, the pointer to the second one-dimensional array is added to the final subscript operand, the value "0". The resulting

pointer addresses the first element of the 6-element array. The last step in evaluating the expression A[2][1][0] is applying the indirection operator to the pointer value. The result is the **int** element at that address.

The indirection operator is applied only when the final pointer value addresses a non-array type. The expression A[2][1] is a perfectly legal reference to the three-dimensional array A; the result of the expression is a pointer value that addresses an array with one dimension.

Member Selection Expressions

A member selection expression has one of two forms:

expression.identifier
expression->identifier

Member selection expressions refer to a member of a structure or union. The value of a member selection expression has the value and type of the selected member.

For the form:

expression.identifier

expression must represent a value of **struct** or **union** type. The *identifier* names a member of the specified structure or union. For the form:

expression->identifier

expression must represent a pointer to a structure or union. The *identifier* names a member of the structure or union pointed to.

The two forms of member selection expressions have a similar effect. In fact, expressions involving "->" are shorthand versions of expressions using "." in cases where the expression before the period consists of the indirection (*) operator (discussed later in this chapter) applied to a pointer value. Thus:

expression->identifier

is equivalent to:

*(*expression).identifier*

when *expression* is a pointer value.

Expressions

Any expression can form an operand of another expression. An expression can consist of any one of the operands described above. An expression can also consist of:

- One or more operands joined by an operator.
- A type cast conversion.
- An expression in parentheses.
- A constant expression.

The general form for each kind of expression is outlined below. Type cast conversions and the unary, binary, ternary and assignment operators are discussed later in this chapter.

Expressions with Operators

An operand prefixed by a unary operator is an expression:

-operand
~operand
!operand
**operand*
&operand
sizeof(*operand*)

An expression can consist of two operands joined by a binary operator (binop):

operand binop operand

An expression can consist of three operands joined by the ternary (? :) operator:

operand ? *operand* :*operand*

An expression can be an assignment expression having one of the following forms:

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

Type Cast Expressions

An expression can consist of a type cast, which has the form:

(*type-name*) *operand*

Type casts are discussed later in this chapter; type names are discussed in Chapter 3.

Expressions in Parentheses

Any expression can be put in parentheses. The parentheses have no effect on the type or value of the enclosed expression. For example, in the expression:

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

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

Although parentheses affect the way operands are grouped in an expression, they cannot guarantee a particular order of evaluation for the expression.

Constant Expressions

A constant expression is an expression that evaluates to a constant. The operands of a constant expression can be integer constants, character constants, enumeration constants, type casts to integral types, and other constant expressions. The operands can be combined using operators, as above; however, certain operators cannot be used in constant expressions. None of the assignment operators can be used; the binary sequential evaluation operator (,) cannot be used; and the unary address of (&) operator may be used only in the circumstances outlined below. Constant expressions in preprocessor directives cannot include **sizeof** expressions and enumeration constants.

A few special allowances for the constant expressions are used to initialize external and static variables. These constant expressions can use floating point constants and type casts to any type. They can also use the unary address of (&) operator with external and static variables, or with external and static arrays subscripted with a constant expression.

Type Conversions

Type conversions take place when:

- A value is assigned to a variable of a different type.
- A value is explicitly cast to another type.
- An operator converts the type of its operand or operands before performing the specified operation.
- A value is passed as an argument to a function.

The rules governing each kind of conversion are outlined below.

Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable receiving the assignment. C allows you to perform conversions by assignment between most types, even when the conversion entails loss of information. The methods of carrying out the conversions depend upon the type, as follows.

Conversions from Signed Integer Types

A signed integer is converted to a shorter signed integer by truncating the high-order bits, and converting to a longer signed integer by sign-extension. Conversion of signed integers to floating point values takes place without loss of information, except that some precision may be lost when a **long** value is converted to a **float.** To convert a signed integer to an unsigned integer, the signed integer is converted to the size of the unsigned integer and the result is interpreted as an unsigned value.

From	To	Method
char	short	sign-extended
char	long	sign-extend to short; convert short to long
char	unsigned char	no change in bit 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 short; convert short to float
char	double	sign-extend to short; convert short to double
short	char	preserve low-order byte
short	long	sign-extend
short	unsigned char	preserve low-order byte
short	unsigned short	no change in bit pattern; high-order bit loses function as sign bit
short	unsigned long	sign-extend to long; convert long to unsigned long
short	float	no change in value
short	double	convert to float; convert float 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	no change in bit pattern; high-order bit loses function as sign bit
long	float	if the long value fits into a float, no change in value occurs; otherwise, some loss of precision occurs
long	double	convert to float; convert float to double

Note: The **int** type is equivalent either to the **short** type or to the **long** type, depending on the implementation. Conversion of an **int** value proceeds as for a **short** or a **long**, whichever is appropriate.

Conversions from Unsigned Integer Types

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits. An unsigned integer is converted to a longer unsigned or signed integer by zero-extending. Unsigned values are converted to floating point values by first converting to a signed integer of the same size, then converting that signed value to a floating point value.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value represented changes if the sign bit is set.

From	To	Method
unsigned char	char	no change in 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 char; convert char to float
unsigned char	float	convert to char; convert char to double
unsigned short	char	preserve low-order byte
unsigned short	short	no change in 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 short; convert short to float
unsigned short	double	convert to short; convert short to double;
unsigned long	char	preserve low-order byte
unsigned long	short	preserve low-order word
unsigned long	long	no change in bit pattern; high-order bit becomes sign bit
unsigned long	unsigned char	preserve low-order byte
unsigned long	unsigned short	preserve low-order word
unsigned long	float	convert to long; convert long to float
unsigned long	double	convert to long; convert long to double

Note: The **unsigned int** type is equivalent either to the **unsigned short** type or to the **unsigned long** type, depending on the implementation. Conversion of an **unsigned int** value proceeds as for an **unsigned short** or an **unsigned long**, whichever is appropriate.

Conversions from Floating Point Types

A **float** converted to a **double** undergoes no change in value; the additional bits are filled with zeros. A **double** converted to a **float** is represented exactly, if possible; if the value is too large to fit into a **float,** precision bits are discarded; if the value is still too large for the **float,** the result is undefined.

A floating point value is converted to an integer value by first converting to **float** size, if necessary, then converting to a **long.** Conversions to other integer types take place as for a **long.** The decimal portion of the floating point 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.

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 fit into a 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	no change in value; additional bits zero-filled
double	char	convert to float; convert float to char;
double	short	convert to float; convert float to short;
double	long	convert to float; convert float to long;
double	unsigned short	convert to float; convert float to unsigned short
double	unsigned long	convert to float; convert float to unsigned long
double	float	if the double value cannot be represented exactly as a float, loss of precision occurs; if the value is still too large to be represented in a float, the result is undefined.

Conversions from Enumeration Type

An **enum** value is an **int** value, by definition of the **enum** type. The size of an **int** is determined by the particular implementation. Conversions to and from an **enum** value proceed as for the **int** type.

Conversions from Structure and Union Types

A structure or union type may only be converted to the same structure or union type.

Conversions from pointer types

A pointer value behaves like an unsigned integer value; the size of a pointer depends on the implementation. Conversions to and from a pointer type proceed as for an unsigned integer of the appropriate size, except that pointers cannot be converted to floating point types.

A pointer to one type of value can be converted to a pointer to a different type. The result can be undefined, however, because of the alignment requirements of different types in storage.

Conversions from Void Type

The **void** type has no value, by definition. Therefore it cannot be converted to any other type, nor can any value be converted to **void** by assignment. However, a value can be explicitly cast to **void** -- see "Type casts", below.

Type Cast Conversions

C allows you to make explicit type conversions by means of a type cast. A type cast has the form:

(*type-name*) *operand*

where *type-name* specified a particular type and *operand* is a value to be converted to the specified type. (Type names are discussed in Chapter 3.)

The conversion of *value* takes place as though it had been assigned to a variable of the named *type*. Thus, the conversion rules for assignments (discussed earlier) apply to type casts as well. The type name **void** can be used in a cast operation, even though a **void** value cannot receive an assignment.

Operator Conversions

The conversions performed by C operators depend on the operator and on the type of the operand or operands. However, many operators perform similar conversions on operands of integral and floating point types. These conversions are known as "arithmetic" conversions because they apply to the types of values ordinarily used in arithmetic. The arithmetic conversions summarized below are called the "usual arithmetic conversions". The discussion of each operator in the next section specifies whether the operator performs the usual arithmetic conversions and any additional conversions.

The usual arithmetic conversions proceed in order as follows:

1. Any operands of **float** type are converted to **double** type.

2. Any operands of **char** or **short** type are converted to **int.**

3. Any operands of **unsigned char** or **unsigned short** type are converted to **unsigned int** type.

4. If one operand is of type **long int,** the other operand is converted to **long int.**

5. If one operand is of type **unsigned long int,** the other operand is converted to **unsigned long int.**

6. If one operand is of type **unsigned int,** the other operand is converted to **unsigned int.**

Function Call Conversions

The usual arithmetic conversions are performed independently on each argument in a function call. Thus, 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.**

In C, arguments are passed by value, not by reference. A copy of each actual argument is made before the function call, and the copies are passed to the function. Thus, changes in the arguments passed to a function are not reflected in the values from which the copies were made.

The types of the formal arguments are not compared against the types of the actual arguments in a function call. If type mismatches occur, they can produce unexpected results.

Operators

C operators take one operand (unary operators), two operands (binary operators), or three operands (the ternary operator).

Unary operators prefix their operand and group right to left. C's unary operators are:

- ~ ! negation operators

* & indirection and address of operators

sizeof size operator

Binary operators group left to right. The binary operators are:

* / % multiplicative operators
+ - additive operators
<< >> shift operators
< > <= >= == != relational operators
& | ^ bitwise operators
&& || logical operators
, sequential evaluation operator

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

Complement Operators

Arithmetic negation (-)

The unary operator - produces the twos complement of its operand. The operand must be an integral or floating point value. An operand of type **char** or **short** is converted to **int.** An **unsigned** integer remains **unsigned;** the negative of an unsigned value is the difference between the unsigned value and the maximum value of an unsigned integer of that size.

Bitwise complement (~)

The unary operator ~ produces the bitwise complement of its operand. The operand must be of integral type. An operand of type **char** or **short int** is converted to **int.** The result has the type of the operand after conversion.

Logical not (!)

The unary operator ! produces a value of 0 if its operand is true (non-zero), and a value of 1 if its operand is false (zero). The resulting value is of type **int.** The operand must be an integral, floating point or pointer value.

Indirection and address of operators

Indirection (*)

The unary operator * yields the value residing at the address specified by its operand. The operand must be a pointer value; the result of the indirection operation is the value to which the operand points. The type of the result is the type to which the operand points. If the pointer value is 0, the result is unpredictable.

Address of (&)

The unary operator & takes the address of its operand. The operand can be any operand that can appear as the left-hand value of an assignment operation. (Assignment operations are discussed later in this chapter.) The result of the & operation is a pointer to the operand; the type addressed by the pointer is the type of the operand.

The **address of** operator cannot be applied to a bit field member of a structure, nor can it be applied an identifier whose storage class is **register,** since the identifier is assumed to refer to a register rather than to addressable memory.

The sizeof Operator

The unary **sizeof** operator enables you to determine the amount of storage associated with an identifier or a type. A **sizeof** expression has the form:

sizeof(*name*)

where *name* is either an identifier or a type name. The type name cannot be **void.** The value of a **sizeof** expression is the amount of storage, in bytes, that is associated with the named identifier or type.

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

When the **sizeof** operator is applied to a structure or union type name, or to an identifier of structure or union type, the result is the actual size in bytes of the structure or union, which can include internal and trailing padding used to align the members of the structure or union on memory boundaries. The result may not correspond to the size calculated by adding up the storage requirements of the members.

Example

```
buffer = calloc(100, sizeof (int) );
```

With the **sizeof** operator you can avoid specifying machine-dependent data sizes in your program. The above example uses the **sizeof** operator to pass the size of an **int** , which varies across machines, as an argument to a function named calloc. The value returned by the function is stored in buffer.

Multiplicative Operators

Multiplication (*)

The binary operator * specifies that its two operands are to be multiplied. The operands must be integral or floating point values, but their types can be different. The multiplication operator performs the usual arithmetic conversions on its operands. The type of the result is the type of the operands after conversion.

Division (/)

The binary operator / specifies that its first operand is to be divided by the second. The operands must be integral or floating point values, but their types may be different. The division operator performs the usual arithmetic conversions. The type of the result is the type of the operands after conversion.

When two integers are divided, the result, if not an integer, is truncated. If both operands are positive or unsigned, the result is truncated toward zero. The direction of truncation when either operand is negative can be either toward or away from zero, depending on the implementation.

Remainder (%)

The result of the binary % operator is the remainder when its first operand is divided by the second. The operands must be integral values, but their types can be different. The remainder operator performs the usual arithmetic conversions. The type of the result is the type of the operands after conversion.

Examples

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

1.  y = x * i;

2.  n = i / j;

3.  n = i % j;
```

In the first example, x is multiplied times i to give the value 20.0. The result has **double** type.

In the second example, 10 is divided by 3. The result is truncated toward zero, yielding the integer value 3.

In the third example, n is assigned the integer remainder 1 when 10 is divided by 3.

Additive Operators

Addition (+)

The binary operator + specifies that its two operands are to be added. The operands can be integral or floating point values, in which case the usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.

One operand can be a pointer and the other an integer. The integer value is converted by multiplying it times the length of the type addressed by the pointer. After conversion, the integer value *i* expresses *i* memory positions of the length specified by the pointer type. When added to the pointer value, the result is a new pointer value expressing the address *i* positions from the original address. The type addressed by the resulting pointer value is the type addressed by the original pointer value.

Subtraction (-)

The binary operator - subtracts its second operand from the first. The operands can be integral or floating point values. The usual arithmetic conversions are performed, and the type of the result is the type of the operands after conversion.

An integer value can be subtracted from a pointer value. As with addition, the integer value is converted with respect to the type addressed by the pointer. The result is the memory address *i* positions before the original address, where *i* is the integer value and each position is the length of the type addressed by the pointer value. The pointer result points to the type addressed by the original pointer value.

Two pointer values can be subtracted, if they point to the same type. The difference between them is converted to a signed integer value by dividing by the length of the type the pointers address. The result represents the number of memory positions of that type between the two addresses.

Pointer and integer combinations

Additive operations involving a pointer and an integer generally yield meaningful results only when the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. The conversion of the integer value to an address offset assumes that only memory positions of the same size lie between the original address and the address plus offset. This assumption is valid for array members. An array is, by definition, a series of values of the same type; its elements reside in contiguous memory locations. Storage of any types except array elements is not guaranteed to be completely filled -- that is, there may be blanks between memory positions, even positions of the same type. Adding to or subtracting from addresses referring to any values but array elements yields unpredictable results.

Similarly, the conversion involved in the subtraction of two pointer values assumes that between the two addresses indicated by the operands lie only values of the same type, with no blanks. Since that assumption is only guaranteed to hold true for two

addresses referring to elements of the same array, subtracting two addresses that refer to anything but members of the same array yields unpredictable results.

Additive operations between pointer and integer values may not be valid on machines with segmented architecture.

Examples

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

1.  px = &x[5] + i;

2.  j = &x[i] - &x[i-2];
```

In the first example, the integer operand i is added to the address of the fifth element of x. The value of i is multiplied by the length of a **float** and added to &x[5] . The resulting pointer value is the address of x[9].

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

Overflow

The conversions performed by the additive operators make no provision for overflow conditions. Information is lost if the result of an additive operation cannot be represented in the type of the operands after conversion. However, the sign of the result is preserved. For instance, if two large **long** values are added and the result does not fit into a **long,** information is lost but the result is guaranteed to be positive. Similarly, an additive operation between two unsigned operands (after conversion) yields an unsigned result.

Shift operators

The binary << and >> operators shift their first operand left (<<) or right (>>) by the number of positions the second operand specifies. Both operands must be integral values. The usual arithmetic conversions are performed; then, if necessary, the second operand is converted to an **int.** The type of the result is the type of the first operand after conversion.

For leftward shifts, the vacated right bits are filled with zeros. The method of filling left bits in the case of a rightward shift depends on the type (after conversion) of the first operand. If it is **unsigned,** vacated left bits will be filled with zeros. Otherwise, vacated left bits are filled with copies of the sign bit.

The result of a shift operation is undefined if the second operand is negative or if it is greater than or equal to the number of bits in the first operand.

Example

```
unsigned int x, y, z;

x = 0x00aa;
y = 0x5500;

z = (x << 8) + (y >> 8);
```

In the above example, x is shifted left by 8 positions and y is shifted right 8 positions. The shifted values are added, giving 0xaa55, and assigned to z.

Relational operators

The binary relational operators test their first operand against the second to determine if the relation specified by the operator holds true. The result of a relational expression is either 1 (if the tested relation holds) or zero (if it doesn't). The type of the result is **int.** The relations tested by the relational operators are the following:

- < First operand less than second operand
- > First operand greater than second operand
- <= First operand less than or equal to second operand
- >= First operand greater than or equal to second operand
- == First operand equal to second operand
- != First operand not equal to second operand

The operands of the relational operators can be of integral or floating point type. For the equality (==) and inequality (!=) operators, one or both operands can be of **enum** type. The usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.

The operands of any relational operator may be two pointers to the same type. For the equality (==) and inequality (!=) operators the result of the comparison reflects whether the two pointers address the same memory location. The result of pointer comparisons involving the other operators (<, >, <=, >=) reflects the relative position of two memory addresses. Since the address of a given value is arbitrary, a comparison between the addresses of two unrelated values is generally meaningless. However, comparisons between the addresses of different elements of the same array can be useful, since array elements are guaranteed to be stored in order from the first element to the last. The address of the first element will be less than the address of the last element.

A pointer value can also be compared for equality (==) or inequality (!=) to the constant value 0. A pointer with a value of 0 does not point to memory location: it is considered a **null** pointer. A pointer value is equal to 0 only if it has been explicitly assigned that value.

Bitwise Operators

The binary operators &, |, and ^ perform bitwise AND, inclusive OR, and exclusive OR operations, respectively. The operands of bitwise operators must be of integral type, but their types can be different. The usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.

Bitwise AND (&)

The bitwise AND operator (&) compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1s, the corresponding bit of the result is set to 1. Otherwise, the corresponding result bit is set to 0.

Bitwise Inclusive OR (|)

The bitwise inclusive OR operator (|) compares each bit of its first operand to the corresponding bit of the second operand. If either of the compared bits is a 1, the corresponding bit of the result is set to 1. Otherwise, both bits are 0s, and the corresponding result bit is set to 0.

Bitwise Exclusive OR (^)

The bitwise exclusive OR operator (^) compares each bit of its first operand to the corresponding bit of the second operand. If one of the compared bits is a 0 and the other bit is a 1, the corresponding bit of the result is set to 1. Otherwise, the corresponding result bit is set to 0.

Examples

```
short i = 0xab00;
short j = 0xabcd;
short n;

1.  n = i & j;

2.  n = i | j;

3.  n = i ¬ j;
```

The result assigned to n in the first example is the same as i, 0xab00. The bitwise inclusive OR in the second example results in the value 0xabcd, while the bitwise exclusive OR in the third example produces 0x00cd.

Logical Operators

The binary operators && and | | are logical AND and OR operators, respectively. The operands of the logical operators must be integral, floating point, or pointer values. These operators do not perform the standard arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to 0. A pointer has a value of 0 only if it has been explicitly set to 0.

The operands of logical AND and OR expressions are evaluated left to right. If the value of the first operand is sufficient to determine the result of the logical operation, the second operand is not evaluated.

Logical AND (&&)

The logical AND operator produces the integer value 1 if both of its operands have non-zero values. If either operand has a value of 0, the result is 0. The type of the result is **int.**

If the first operand of a logical AND operation has a value of 0, the second operand is not evaluated, since the result of the operation is 0 when either operand is 0.

Logical OR (| |)

The logical OR operator performs an inclusive OR on its operands. It produces the integer value 0 if both of its operands have 0 values. If either operand has a non-zero value, the result of the operation is 1. The type of the result is **int**.

If the first operand of a logical OR operation has a non-zero value, the second operand is not evaluated, since the result of the operation is 1 when either operand is non-zero

Examples

```
int x, y;

1.  if (x < y && y < z)
         printf ("x is less than z\n");

2.  if (x == y || x == z)
         printf ("x is equal to either y or z\n");
```

In the first 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, "y < z" is not evaluated and nothing is printed.

In the second example, a message is printed if x is equal to either y or z. If x is equal to y, "x == z" is not evaluated.

Sequential Evaluation Operator (,)

The binary operator (,) evaluates its two operands sequentially from left to right. The result of the operation has the value and type of the right operand. The types of the operands are unrestricted. No conversions are performed.

The sequential evaluation operator evaluates two expressions in contexts that allow only one expression to appear. For instance, the optional third expression of a **for** statement (discussed in Chapter 5) might use the , operator to alter the values of two variables instead of one:

```
for ( i = j = 1; i + j < 20; i += i, j--) ;
```

When the third expression, i += i, j--, is evaluated, each operand is evaluated independently. The left operand, i += i, is evaluated first, then j-- is evaluated.

The comma character in other contexts acts as a separator-- for instance, between the arguments to a function. If you wish to use the sequential evaluation operator in situations where it could be misinterpreted, you must use parentheses to group expressions in an unambiguous way. For example, the function call:

```
f (x, y + 2, z);
```

passes three arguments to the called function: x, y + 2, and z, while:

```
f ( (x, y + 2), z);
```

passes two arguments. 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.

The Conditional Operator (?:)

C has one ternary operator, the conditional (? :) operator. Its form is:

operand1 ? operand2 : operand3

The first operand is evaluated in terms of its equivalence to 0; it must be an integral, floating point, or pointer value. If the first operand has a non-zero value, the result of the expression is the value of the second operand. If the first operand evaluates to 0, the result of the expression is the value of the third operand.

The type of the result depends on the types of the second and third operands, as follows:

- Both the second and third operands can be of integral or floating point type. The usual arithmetic conversions are performed, and the type of the result is the type of the operands after conversion.

- Both the second and third operands may be of the same structure, union, or pointer type. The type of the result is the same structure, union, or pointer type.

- One of the second and third operands can be a pointer and the other a constant expression with the value 0. The type of the result is the pointer type.

The following is an example of a conditional expression:

```
i < 0 ? j = -i : j = i;
```

If i is less than 0, the value -i is assigned to j; if i is greater than or equal to 0, i is assigned to j.

Assignment Operators

C's assignment operators are:

++	Unary increment operator
--	Unary decrement operator
=	Simple assignment operator
*=	Multiplication assignment operator
/=	Division assignment operator
%=	Remainder assignment operator
+=	Addition assignment operator
-=	Subtraction assignment operator
<<=	Left shift assignment operator
>>=	Right shift assignment operator
&=	Bitwise AND assignment operator
|=	Bitwise inclusive OR assignment operator
^=	Bitwise exclusive OR assignment operator

An assignment operation specifies that the value of the right-hand operand is to be deposited in (assigned to) the storage location named by the left-hand operand. Thus, the left-hand operand of an assignment operation must be an expression referring to a memory location. Expressions that refer to memory locations are called lvalue expressions. A variable name is such an expression: the name of the variable denotes a storage location, while the value of the variable is the value residing at that location. Constant values or expressions that refer to constant values (such as array and function identifiers and string expressions) are not lvalue expressions.

The C expressions that can be lvalue expressions are:

- Identifiers of character, integer, floating point, pointer, enumeration, structure, or union type.
- Subscript ([ ]) expressions, except when a subscript expression evaluates to a pointer to an array.
- Member selection expressions (-> and .), if the selected member is one of the above expressions.
- Unary indirection (*) expressions.
- The lvalue expression in parentheses.

Unary Increment and Decrement

The unary assignment operators ++ and -- increment and decrement their operand, respectively. The effect of an increment or decrement operation is to perform addition or subtraction between the integer value 1 (which may be converted) and the operand, then assign the new value back to the operand. The operand must be an integral, floating point, or pointer value, and must be an lvalue expression.

Operands of integral or floating point type are incremented or decremented by the integer value 1. The type of the result is the type of the operand. An operand of pointer type is incremented or decremented with regard to the size of the type it addresses. An incremented pointer points to the next memory location; a decremented pointer points to the previous memory location.

A ++ or -- operator can appear either before or after its operand. When the operator prefixes its operand, the result of the expression is the incremented or decremented value of the operand. When the operator postfixes its operand, the immediate result of the expression is the value of the operand *before* it is incremented or decremented. After that result is noted in context, the operand is incremented or decremented.

Examples

1.
```
if (c > 0)
      pos++;
```

2.
```
if (line[--i] != '\n')
      continue;
```

In the first example, the variable pos is incremented by one if c is greater than zero. In this case, the increment operator can be placed either before or after pos without changing the result.

In the second example, the variable i is decremented before the *ith* element of line is compared to the character '\n'.

Simple Assignment

The simple assignment operator (=) performs assignment. The right operand is assigned to the left operand; the conversion rules for assignment apply.

Example

```
double x;
int y;

x = y;
```

The value of y is converted to **double** type and assigned to x.

Compound Assignment

The compound assignment operators consist of the simple assignment operator combined with another binary operator. Compound assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. A compound assignment expression such as:

expression1 += *expression2*

may 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 in the expanded version *expression1* is evaluated twice: in the addition operation and in the assignment operation.

Each compound assignment operator performs the conversions that the corresponding binary operator performs, and restricts the types of its operands accordingly. The result of a compound assignment operation has the value and type of the left operand.

Example

```
#define MASK    0xffff

n |= MASK;
```

In this example a bitwise inclusive OR operation is performed on n and MASK and the result is assigned to n. The manifest constant MASK is defined with a preprocessor directive, discussed in Chapter 7.

Precedence

The precedence and associativity of C operators affect how operands of an expression are grouped and evaluated. An operator's precedence is meaningful only in the presence of other operators having higher or lower precedence. Expressions involving higher precedence operators are evaluated first.

Operands consisting of a constant, an identifier, a string, a function call, a subscript expression, a member selection expression, or a parenthesized expression have higher precedence than any operator, and group left to right. Type-cast conversions have the same precedence and grouping as the unary operators.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator -- that is, either right to left or left to right. The result of expressions involving multiple occurrences of multiplication (*), addition (+), or binary bitwise (&, | ,^) operators at the same level is indifferent to the direction of evaluation. The compiler is free to evaluate such expressions in any order, even when parentheses in the expression appear to specify a particular order.

The following list summarizes the precedence and associativity of C operators. The operators are listed in order of precedence from the highest to the lowest. Where several operators appear together on a single line or in the same box, they have equal precedence.

Kind	Operator (highest precedence first)		Grouping
expression	( ) [ ] -> .	Expression operators	Left to right
unary	! ~ ++ -- - (type) * & sizeof	Unary operators	Right to left
binary	* / %	Multiplicative operators	Left to right
binary	+ -	Additive operators	Left to right
binary	<< >>	Shift operators	Left to right
binary	< <= > >=	Relational operators	Left to right
binary	== !=	Equality operators	Left to right
binary	&	Bitwise AND	Left to right
binary	^	Bitwise exclusive OR	Left to right
binary	|	Bitwise inclusive OR	Left to right
binary	&&	Logical AND	Left to right
binary	||	Logical OR	Left to right
ternary	? :	Conditional operators	Right to left
binary	= *= /= %= += -= <<= >>= &= != ^=	Simple and compound assignment	Right to left
binary	,	Sequential evaluation operator	Left to right

Only the sequential evaluation operator (,) and the logical AND and OR operators (&& and | |) guarantee a particular order of evaluation for the operands. The sequential evaluation operator (,) always evaluates its operands from left to right.

The logical operators will also evaluate their operands left to right. However, the logical operators evaluate the minimum number of operands necessary to determine the result of the expression. This means that 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 (non-zero). Thus, y is not incremented when x is false (zero).

Expression	Default Grouping
1. a & b		c	(a & b)		c
2. a = b		c	a = (b		c)
3. q && r		s- -	(q && r)		s- -
4. p == 0 ? p += 1 : p += 2	(p == 0 ? p += 1 : p) += 2

NOTE: Example 4 produces an error.

In the first example, 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 example, 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. Notice that the value assigned to "a" is either zero or one.

The third example 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 non-zero value, s- - is not evaluated. This means that " s" is not decremented. To correct this problem, s- - should appear as the first operand of the expression, or should be decremented in a separate operation.

The fourth example shows an illegal expression that produces a program error. In this example, the equality operator (==) has the highest precedence, so p == 0 is grouped as an operand. The ternary operator (? :) has the next highest precedence. The first operand of the ternary operand is p == 0 and the second operand is p += 1. Notice that the compiler considers the expression between the question mark (?) and the colon (:) to be the second operand and evaluates it as a unit, even though the compound addition operator (+=) has lower precedence than the ternary operator. However, the compiler considers the last operand of the ternary operator to be " p" rather than p += 2 .

This occurrence of p binds more closely to the ternary operator than it does to the compound assignment operator. When the last p is grouped with the ternary operator, a syntax error occurs because += 2 does not have a left-hand operand.

This example can be corrected and clarified through the use of parentheses, as shown below.

```
(p == 0) ? (p += 1) : (p += 2)
```

Side Effects

Side effects are changes in the state of the machine that take place as a result of evaluating an expression. Thus, side effects occur in any expression involving assignment. The order of evaluation of side effects is implementation-dependent.

Assignment expressions must be used with care. For example, side effects occur in the following function call:

```
add ( i + 1, i = j + 2)
```

The arguments of a function call can be evaluated in any order; thus, "i + 1" may be evaluated before "i = j + 2", or vice versa, with different results in each case.

Unary increment and decrement operations involve assignments and can cause side effects, as shown in the following example:

```
d=0;
a = b++=c++=d++;
```

The value of *a* is unpredictable. The initial value of *d*(zero) could be assigned to *c*, then to *b*, then to *a* before any of the variables a re incremented. In this case, *a* would be equal to zero.

A second method of evaluating this expression begins by evaluating the operand "c++=d++". The initial value of *d*(zero) is assigned to *c*, then both *d* and *c* are incremented. Next, the incremented value of *c*(1) is assigned to *b* and *b* is incremented. Finally, the incremented value of *b* is assigned to *a*. In this case, the final value of *a* is 2.

Since the C language does not define the order of evaluation of side effects, both of these evaluation methods are correct and either can be implemented. Statements that depend on a particular order of evaluation for side effects produce non-portable and unclear code.

Chapter 5. Statements

Contents

Introduction **5-3**

Break statement **5-5**
- Syntax 5-5
- Execution 5-5
- Example: 5-5
- Exiting from Nested Statements 5-6

Compound statement **5-7**
- Syntax 5-7
- Execution 5-7
- Example: 5-8
- Labeling Statements 5-8

Continue statement **5-9**
- Syntax 5-9
- Execution 5-9
- Example: 5-9

Do statement **5-10**
- Syntax 5-10
- Execution 5-10
- Example: 5-10

Expression statement **5-11**
- Syntax 5-11
- Execution 5-11
- Example: 5-11
- Assignments and Function Calls 5-11

For statement **5-12**
- Syntax 5-12
- Execution 5-12
- Example: 5-13

Goto and labeled statements **5-14**
Syntax 5-14
Execution 5-14
Example: 5-14
Forming labels 5-15

If statement **5-15**
Syntax 5-15
Execution 5-15
Example: 5-15
Nesting 5-16

Null statement **5-17**
Syntax 5-17
Execution 5-17
Example: 5-17
Labeling a null statement 5-17

Return statement **5-18**
Syntax 5-18
Execution 5-18
Example: 5-18
Omitting the return statement 5-19

Switch statement **5-20**
Syntax 5-20
Execution 5-20
Examples: 5-22
Labeling statements 5-23

While statement **5-24**
Syntax 5-24
Execution 5-24
Example: 5-24

Introduction

The statements of a C program control the flow of program execution. In C, as in other programming languages, there are several kinds of statements. This chapter describes C statements in the following order:

break statement
compound statement
continue statement
do statement
expression statement
for statement
goto statement
if statement
null statement
return statement
switch statement
while statement

C statements consist of keywords, expressions, and other statements. The keywords that appear in C statements are:

break
case
continue
default
do
else
for
goto
if
return
switch
while

The expressions in C statements are the expressions discussed in Chapter 4. Statements appearing within C statements can be any of the statements discussed in this chapter. A statement that is a component of another statement is called the "body" of the enclosing statement.

C statements end with a semicolon. The only exception to this rule is the compound statement, which is delimited by braces. The right brace serves as the terminator for the compound statement.

Any C statement can be prefixed with an identifying label consisting of a name and a colon. Name labels are recognized only by the **goto** statement and are therefore discussed with the **goto** statement.

The effect of a C program's execution is that of the execution of its statements in order of their appearance in the program, except where a statement explicitly transfers control to another location. To produce efficient code, the compiler can rearrange the actual order of statement execution in any way that does not change the program's effect.

Break statement

Syntax

break ;

Execution

The **break** statement terminates the execution of the **do** , **for** , **switch** or **while** statement in which it appears. Control passes to the next statement in the program. If the **break** statement appears in a nested statement, only the smallest enclosing **do, for, switch** or **while** terminates with the **break** statement's execution.

Example:

```
switch (i) {
     case 1:
            f(i);
            break;
     case 2:
            g(i);
            break;
     default:
            h(i);
}
```

In the example, **break** is used to exit from the **switch** after one **case** labeled statement is executed. If i equals 1, f(i) is executed, then the **break** statement following it is executed, and control passes out of the **switch**, bypassing the remaining lines of the **switch** body. If i equals 2, g(i) is executed, then the **break** statement following it is executed, and control passes out of the **switch,** bypassing the **default** labeled statement. If i does not equal either 1 or 2, the **default** labeled statement is executed. Since the **default**

labeled statement is the last statement in the **switch** body, control subsequently passes to the next statement in the program without the execution of a **break** statement.

Exiting from Nested Statements

The **break** statement terminates only the **do, for, switch** or **while** statement which immediately encloses it. Within a nested statement this has the effect of transferring control to the level of the next outermost statement. To transfer control out of the nested structure altogether, you may want to use a **return** or **goto** statement rather than a **break**.

Compound statement

Syntax

```
{
  [ declaration ]
  [ declaration ]
    .
    .
    .
  statement
  [ statement ]
    .
    .
    .
}
```

Execution

The effect of a compound statement's execution is that of the execution of its statements in order of their appearance, except where a statement explicitly transfers control to another location. To produce efficient code, the compiler can rearrange the actual order of execution in any way that does not alter the statement's effect.

Example:

A compound statement is typically used as the body of another statement such as the **if** statement:

```
if (i > 0) {
        A[i] = x;
        x++;
        i--;
}
```

If i is greater than 0, the compound statement is executed: first x is assigned to A[i], then x is incremented, then i is decremented. If i is less than or equal to 0, none of the statements in the compound statement are executed.

Labeling Statements

Like other C statements, each of the statements in a compound statement can carry a label. Transfer into the compound statement by means of a **goto** is therefore possible. However, it is dangerous when the compound statement includes declarations that initialize **auto** or **register** variables. Declarations in a compound statement precede the executable statements; transferring directly to an executable statement within the compound statement bypasses the initializations, with unpredictable results.

Continue statement

Syntax

continue;

Execution

The **continue** statement terminates the current iteration of the **do, for,** or **while** statement body in which it appears. If the **continue** statement is executed within a **do** or a **while** statement, execution resumes starting with the re-evaluation of the **do** or **while** statement's expression. If the **continue** statement is executed within a **for** statement, execution resumes with the evaluation of the **for** statement's third expression, if present, and proceeds with the evaluation of the second expression (if present) and subsequent termination or reiteration of the statement body.

Example:

```
while (i-- > 0) {
            x = f(i);
            if (x == 1)
                   continue;
            y = x * x;
}
```

In the example, if i is greater than 0, execution of the statement body begins. First, f(i) is assigned to x. Then, if x is equal to 1, the **continue** statement executes. The last statement in the body is ignored and execution resumes with the evaluation of the expression i-- > 0. If x does not equal 1, the **continue** statement is not executed. Control passes to the next statement in the body, y = x * x; following its execution, the **while** statement is executed again, starting with the evaluation of i > 0.

Do statement

Syntax

```
do
   statement
while (expression);
```

Execution

The **do** statement executes its statement body one or more times, depending on the value of its expression. First, the statement body is executed. Then the expression is evaluated. If the expression is false (0), the **do** statement terminates and control passes to the next statement in the program. Notice that the body of a **do** statement is executed at least once, even if the expression is initially false. If the expression is true (non-0), the statement body re-executes and the expression is tested again. The **do** statement continues to execute until the expression becomes false.

The **do** statement can also terminate with the execution of a **break, goto,** or **return** statement within the statement body.

Example:

```
do {
        y = f(x);
        x--;
} while (x > 0);
```

In the example, the two statements y = f(x) and x-- are executed. Then x > 0 is evaluated. If x is greater than 0, the statement body is executed again, and x > 0 is re-evaluated. The statement body is executed repeatedly as long as x remains greater than 0. When x becomes 0 or negative, x > 0 is false, and execution of the **do** statement terminates.

Expression statement

Syntax

expression ;

Execution

The expression is evaluated.

Example:

```
x = 3.5;
```

The evaluation of x = 3.5 results in the assignment of the value 3.5 to the variable x .

Assignments and Function Calls

In C, assignments are expressions; the value of the expression is the value being assigned (sometimes called the right-hand value). Most expression statements are assignments, for the simple reason that you typically want to store the value of the expression for later use.

Function calls are expression statements. To capture a value returned by a function, the expression statement must incorporate an assignment. A function whose return type is **void** returns no value and no assignment is necessary.

For statement

Syntax

for ([*expression1*];[*expression2*];[*expression3*])
statement;

Execution

The **for** statement initializes and modifies the value of a variable or variables used in the repeated execution of the statement body. *Expression1* is the initializing expression; the value of *expression2* determines whether the statement body is executed; and *expression3* allows for modification of values used in the statement body. The first step in the execution of the **for** statement is the evaluation of the first expression, if it is present. Next, the second expression is evaluated. There are three possible results:

- The second expression is true (non-zero): first, the statement body is executed; then the third expression, if present, is evaluated; then the process begins again with the evaluation of the second expression.

- The second expression is omitted: the second expression is considered true; execution proceeds exactly as above. A **for** statement lacking a second expression terminates only upon the execution of a **break**, **goto**, or **return** within the statement body.

- The second expression is false: execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement can also terminate with the execution of a **break**, **return**, or **goto** statement within the statement body.

Example:

```
for (i = space = tab = 0; i < MAX; i++) {
        if (line[i] == '\0x20')
                space++;
        if (line[i] == '\t') {
                tab++;
                line[i] = '\0x20';
        }
}
```

The above example counts space (\0x20) 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 zero. Then i is compared to the constant MAX; if i is less than MAX, the statement body is executed. Depending on the value of line[i], the body of one or neither of the **if** statements is executed. Then i is incremented and tested against MAX. The statement body is executed repeatedly as long as i is less than MAX.

Goto and labeled statements

Syntax

goto *name*;

.
.
.

name: *statement*;

Execution

The **goto** statement transfers control directly to the statement specified by *name*. The labeled statement is executed immnediately after the execution of the **goto** statement. An error results from the execution of the **goto** if no statement with the specified label resides in the same function or if the same label name appears before more than one statement in the function.

A statement label is meaningful only to a **goto** statement. When a labeled statement is encountered in any other context it is executed without regard to its label.

Example:

```
if (errorcode > 0)
        goto exit;
   .
   .
   .
exit:
    return (errorcode);
```

Forming labels

Label names are constructed following the same rules that govern the construction of identifiers. Each statement label must be distinct from the other statement labels within the same function.

If statement

Syntax

if (expression)
    statement1;

[else
    statement2;]

Execution

The **if** statement executes its statement body selectively, depending on the value of its expression. First, the expression is evaluated. If the expression is true (non-zero), the statement immediately following the expression executes. If the expression is false (zero) and there is no **else** clause, the trailing statement is ignored, and control passes to the next statement in the program. If the expression is false and there is an **else** clause, the statement following the **else** is executed.

Example:

```
if (i > 0)
    y = x/i;
else
    x = i;
```

In the example, the value x/i is assigned to y if i is greater than 0; if i is less than or equal to 0, i is assigned to x. Notice that both the **if** and the **else** clauses end with semicolons.

Nesting

An **if** statement can appear in either the **if** clause or the **else** clause of another **if** statement. In some cases, nested **if** statements can produce ambiguity. If you freely nest **if** statements with **else** clauses inside **if** statements without **else** clauses (or vice versa), you may find it difficult to determine which **if** is the antecedent of a particular **else**. Using braces to group the statements and clauses into compound statements help clarify your intent. In the absence of braces, C resolves ambiguities by pairing each **else** with the most recent **if** lacking an **else**.

Without braces

```
if (i > 0)
        if (j > i)
                x = j;
else
                x = i;
```

With braces

```
if (i > 0) {
        if (j > i)
                x = j;
     }
else
   x = i;
```

In the first version, the **else** is associated with the second **if**. If i is less than or equal to 0, no part of the second **if** statement executes, and nothing is assigned to x. If i is greater than 0, the second **if** statement executes. The variable j is assigned to x if j is greater than i. If j is less than or equal to i, i is assigned to x.

In the second version, the braces surrounding the second **if** statement force the **else** to be considered part of the first **if** statement. If i is less than or equal to 0, the **else** clause of the first **if** is executed and i is assigned to x. If i is greater than 0, j is assigned to x only if j is greater than i. If i is greater than 0 and j is less than or equal to i, nothing is assigned to x.

Null statement

Syntax

```
;
```

Execution

A null statement is a statement containing only a semicolon that may be used wherever a statement is expected. Nothing happens when a null statement executes.

Example:

```
for (i = 0; i < 10; A[i++] = 0)
    ;
```

The syntax of statements such as **do, for, if** and **while** requires that an executable statement appear as the body of the statement. The null statement allows you to satisfy the syntax requirement neatly in cases where you have no need for a substantive statement body. In the example above, the **for** statement initializes the first 10 elements of A to 0. The third expression accomplishes the purpose of the **for** statement by setting A[i], to 0, then incrementing i. A null statement follows the expression list of the **for** statement, since no further statements are necessary.

Labeling a null statement

The null statement, like any other statement, can be prefixed by an identifying label. If you wish to label an item that is not a statement, such as the closing brace of a compound statement, you can insert and label a null statement immediately before the item to get the same effect.

Return statement

Syntax

return [*expression*] ;

Execution

The **return** statement returns control from a function to the function that called it. Execution of a **return** statement terminates the execution of the function in which it appears. The calling function resumes execution at the point just after the call. If the **return** statement includes the optional expression, the value of the expression is returned to the calling function.

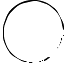
Example:

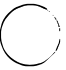
```
main()
{
           .
           .
           .
       y = sq(x);
           .
           .
           .
}
sq(x)
int x;
   {
   return (x * x);
   }
```

The statement y = sq(x) in **main** calls the function “sq” with the argument x. The function body contains only a **return** statement, which returns the value x * x to the **main** function. Execution of **main** resumes at the assignment of the return value to y.

Omitting the return statement

If no **return** statement is encountered before the last statement of a function executes, control automatically returns to the calling function at the point just after the call. The return value of the called function is undefined.





Switch statement

Syntax

```
switch ( expression ) {
    [ declaration ]
  .
  .
  .
          case constant-expression :
          statement
        [case constant-expression :
          statement]
  .
  .
  .
        [default :
          statement ]
        [case constant-expression :
          statement]
  .
  .
  .
}
```

Execution

The **switch** statement transfers control to a statement within its body. The statement receiving control is the statement whose **case** *constant-expression* matches the parenthesized *expression* . Execution of the statement body begins at the selected statement and proceeds through the end of the body or until a statement transfers control out of the body.

The **default** statement is executed if no **case** *constant-expression* is equal to the **switch** *expression* . If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body is executed.

The **switch** expression must be an integral or **enum** value. 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 the starting point for execution of the statement body. Once execution of the statement body begins, **case** and **default** labels do not affect control flow. All statements appearing between the statement where execution starts and the end of the body are executed regardless of their labels, unless a statement transfers control out of the body entirely.

Declarations may appear at the head of the compound statement forming the **switch** body, but initializations included in the declarations are not performed. The effect of the **switch** statement is to transfer control directly to an executable statement within the body, bypassing the lines that contain initializations.

Examples:

1.

```
switch (c) {
        case 'A':
                capa++;
        case 'a':
                lettera++;
        default :
                total++;
        }
```

2.

```
switch (i) {
        case -1:
                n++;
                break;
        case 0 :
                z++;
                break;
        case 1 :
                p++;
                break;
                }
```

In the first example, capa, lettera, and total are incremented if c is equal to the character "A". Lettera and total are incremented if c equals "a". Only total is incremented if c is not equal to a or A .

The **break** statement is used in the second example to force an exit from the **switch** after one statement in the body is executed. If i is equal to -1, n is incremented and the **break** statement executes. Control passes out of the **switch** body, bypassing the remaining statements. Similarly, if i is equal to zero, z is incremented, and the following **break** statement transfers control out of the **switch** body. For consistency, the **break** statement is included after the last **case** labeled statement in the body, although it is not strictly necessary.

Labeling statements

A statement can carry multiple **case** labels. The syntax is:

case *constant-expression* :

.
.
.

case *constant-expression* : *statement*

Although any statement within the body of the **switch** statement can be labeled, no statement is required to carry a label. Statements without labels can be freely intermingled with labeled statements. Keep in mind, however, that once the **switch** statement passes control to a statement within the body, all succeeding statements in the block are executed, regardless of their labels.

While statement

Syntax

while *(expression)* statement;

Execution

The **while** statement executes its statement body 0 or more times, depending on the value of its expression. The first step in the **while** statement's execution is the evaluation of the expression. If the expression is false (0), the **while** statement stops executing and control passes to the next statement in the program. The body is never executed if the expression is initially false. If the expression is true (non-zero), the body of the statement executes. Following each execution of the statement body the expression is re-evaluated; if it is true, the body re-executes and the process repeats, starting with the re-evaluation of the expression.

The **while** statement can also terminate with the execution of a **break, goto,** or **return** within the statement body.

Example:

```
while (i >=  0) {
     A[i] = B[i];
     i--;
}
```

If i is greater than or equal to 0, B[i] is assigned to A[i] and i is decremented. As long as i is greater than or equal to 0, the process repeats. When i reaches or falls below 0, execution of the **while** statement terminates.

Chapter 6. Functions

Contents

Introduction 6-3
Function Definition 6-4
Return Value Type 6-5
Formal Parameters 6-8
Function Body 6-10
Function Declarations 6-11
Static Functions 6-13
Function Calls 6-14
Actual Parameters 6-15
Fundamental Types 6-16
Arrays 6-17
Structures and Unions 6-18
Pointers 6-19
Function Pointers 6-21
Recursive Functions 6-23

Introduction

A function is an independent collection of statements and variables that performs a specific task. Every program has at least one main function and can have any number of other functions. All functions within a program must be defined. An explicit function definition defines the name of the function and the statements that define its action. A function declaration implicitly or explicitly defines just the name and return type of a function whose explicit definition is given later in the source file or in another source file.

The following sections explain how to declare functions and how to execute a function with a function call. In particular, the sections describe:

- Function Definitions
- Return Values
- Formal Parameters
- Function Body
- Function Declarations
- Static Functions
- Function Calls
- Actual Parameters
- Recursive Functions

Function Definition

A function definition defines the name, statements, and variables of a function. It also defines the function's formal parameters, scope, and return value type. A function definition has the form:

[*storage-class*][*type-specifier*] *declarator*(*parameter-list*)
parameter-declarations
function-body

where *storage-class* defines the function's storage class (**static** or **extern**), *type-specifier* is the function's return value type, *declarator* is the function's name, a unique identifier, *parameter-list* is a list of formal parameters to be used by the function, *parameter-declarations* are declarations of the formal parameters, and *function-body* is a compound statement containing the statements and local variable declarations of the function. For example, the following is a complete function definition:

```
int strfind(s, c)
char s[ ], c;
{
        int i;

        for (i=0; s[i] != c && s[i] != 0; i++)
                ;
        return (i);
}
```

This function searches for a given letter in a string and returns the position of that letter within the string. The function's name is strfind. It returns a value with **int** type, and accepts two parameters "s" and "c". These parameters are declared at the beginning of the function body as an array of characters and a character. The compound statement in the function body contains a declaration for the integer variable i, used to keep the current position. The **for** statement performs the search and the **return** statement returns the result.

The following sections describe each part of the function declaration in detail.

Return Value Type

The return value type defines the size and type of value to be returned by a function. The return value type can be explicitly declared with the function name at the beginning of the function declaration. The type declaration has the form:

[*type-specifier*] *declarator*

where *type-specifier* defines the type, and *declarator* defines the function's name and other attributes. The type can be any fundamental or aggregate type. If no type is given **int** is assumed. For example, in the following program fragment, the function add has the implied **int** type but addd has **long**.

```
add (x, y)        /* int return type by default */
int x, y;
{
        return (x+y);
}

long addd(x, y)   /* long return type */
long x, y;
{
        return (x+y);
}
```

There are few restrictions on the type of value a function can return. Although a function can only return a pointer to an array, it can return an entire structure or union.

For example, in the following program fragment, the function sortrec chooses and returns an entire record.

```
typedef struct {
            char name[20];
            int id;
            long class;
} record;

record sortrec ( a, b )      /* Return type is "record" */
record a, b;
{
   return ( (a.id < b.id) ? a : b );
}
```

To return an array, the function must return a pointer to the first element of the array. This means the return value type must declare a pointer. For example, in the following program fragment, the return value of the function smallstr is a pointer to an array of characters.

```
char *smallstr(s1, s2)      /* Return type is pointer to char */
char s1[ ], s2[ ];
{
       int i;
       i=0;
       while ( s1[i] != 0 && s2[i] != 0 )
         i++;
       if ( s1[i] == 0 )
               return (s1);
       else
               return (s2);
}
```

In this example, the return value type is defined as:

```
char *smallstr
```

This means the function smallstr returns a pointer to a character variable in an array of characters.

A function can also return pointers to other types of variables. For example, the following function returns a pointer to a structure.

```
typedef struct {
            char name[20];
            int id;
            long class;
    } record;
record *build(name, id, class)/*Return type is pointer to record*/
char name[ ];
int id;
long class;
{
    int i;
    static record rec;

    i=0;
    while ( i<20 && name[i] != 0 ) {
          rec.name[i] = name[i];
          i++ ;
}
rec.id = id;
rec.class = class;

return ( &rec );
}
```

In this example, the return value type is defined to be:

```
record *build
```

where record has been defined to be a type name defined by the **typedef** declaration. This function returns a pointer to a structure having the same members as record.

A function's return value type is only used when the function returns a value. The function must contain a **return** statement with an expression. The function evaluates the expression, converts it to the return value type (if necessary), and returns it to the point of call. If the function does not contain a **return** statement or the statement has no expression, then the return value type is not used.

Formal Parameters

Formal parameters are variables that receive values passed to a function by a function call. The formal parameters must be explicitly defined in a parameter list at the beginning of the function declaration and can be declared in the declaration list at the beginning of the function body.

The parameter list defines the names of the parameters and the order in which they are assigned values from the function call. The parameter-list has the form:

([*identifier*] [,*identifier*] . . .)

where *identifier* is a unique identifier that names the parameter. There must be one identifier for each value to be passed to the function. If no identifiers are given, no values can be passed to the function. The enclosing parentheses are required.

The declaration list declares the type and size of value each parameter can receive. There must be one declaration for each identifier in the parameter list. Parameter declarations have the same form as other variable declarations (see Chapter 3, "Declarations"). Note that a parameter can only have **auto** or **register** storage class. If no storage class is given, automatic storage is assumed. If no type is given, **int** is assumed. A type is required if no storage class is given. Formal parameters can be declared in any order. If a formal parameter is given in the parameter list but no declaration is given, the parameter is assumed to have **int** type.

For example, in the following function fragment, one formal parameter **x** is declared.

```
truth (x)  /* Single parameter x. */
int x;
{
          return ( x&&1 );
}
```

In the next example, the formal parameters x and y are declared together.

```
add (x,y) /* Two parameters x and y. */
int x, y;
{
        return ( x+y );
}
```

There are few restrictions on the type of formal parameters that can be declared. A formal parameter can be an array, structure, or union as well as any variable with a fundamental type. For example, in the following program fragment, the function "match" has both a structure pointer and a pointer to an array of characters as formal parameters.

```
match ( r, n )
struct record *r;
char n[ ];
{
        int i = 0;

        while ( r->name[i] == n[i] ){
                if ( r->name == 0 ) return (r->id);
                i++;
                 }
        return (0);
}
```

A formal parameter cannot be a function, but it can be a pointer to a function.

Do not declare ordinary local variables in the declaration list. This list is reserved for the formal parameters defined in the parameter list. Declare local variables within the function body.

Function Body

The function body is simply a compound statement. The compound statement contains the statements that define the function's action and can also contain declarations for variables used by these statements. The function body has the same form as compound statements described in Chapter 5, "Statements." The following example illustrates a function body.

```
wait (x)
int x;
{
         int i;

         for (i=0; i<x; i++)
              ;
}
```

All variables declared in the function body have automatic storage type unless otherwise specified. Their scope is restricted to the function body.

When the function is called, storage space for the automatic variables is created and any local initializations are performed, then execution control passes to the first statement in the compound statement. Execution continues sequentially until a **return** statement or the end of the function body is encountered. Control then passes back to the point of call. A **return** statement is required only if the function must return a value. The end of the function body can return control, but it cannot return a value.

Function Declarations

Function declarations define the name, return type, and storage class of a given function. There are two types of function declaration: forward declarations and implicit declarations.

Forward declarations counteract the compiler's implicit declaration of a function. A forward declaration has the form described for function declarations in Chapter 3, "Declarations." For example, in the following program fragment the forward declaration:

```
long addd();
```

defines the return type of the function addd to be **long**, overriding the compiler's implicit type.

```
long addd();      /* Forward declaration */

main ()
{
     long x=10, y=20;

     addd(x,y);    /* Function call to addd */
}
long addd(a,b)   /* Function definition */
long a,b;
{
       return (a+b);
}
```

A forward declaration of a function can appear in the function body of another function. This is allowed since no statements or formal parameters are actually defined in the forward declaration. Functions defined in this way are still given external storage class; automatic storage class for functions is not allowed.

An implicit declaration occurs whenever a function call used in an expression or statement corresponds to no previously defined or declared function. The C compiler implicitly declares the

function to have **int** return type. For example, in the following program fragment, the function "add" is implicitly declared since the function call appears before its formal declaration.

```
main ()
{
        int x=1,y=2;

        add(x,y);
}

add (a, b)
int a, b;
{
        return ( a+b );
}
```

If a function has been implicitly declared, the formal declaration must match the implied return type, otherwise an error results.

Static Functions

A function is normally known in the source file in which it is defined and in source files in which it is declared to be external (see the section "Visibility and Scope"). Functions default to extern storage class. If desired, a function can be restricted to a single source file by explicitly declaring the function with the **static** storage class. Such a declaration causes the function's name to be known only to the other functions in the same source file. For example, in the following program fragment the function "add" is known to the **main** function only. Other functions (in other source files) cannot access "add".

```
static add();

main ()
{
        int x=1,y=2;

        add (x+y);
}

static add ( a, b)
int a, b;
{
    return ( a+b );
}
```

A function cannot have automatic or register storage class.

Function Calls

A function call is any C expression that passes control and possibly one or more actual parameters to a function. A function call has the form described in Chapter 4, "Expressions and Assignments."

When encountered in a program, the function call evaluates the expressions in the expression list, assigns these values to the given function's formal parameters, then passes execution control to the first statement in the function.

Care must be taken when making assignments in the expression list since the order in which the arguments are evaluated is not defined.

Examples

1.
```
add ( 2, i+3 )
```

   This example passes "2" and the value of the expression i+3 to the formal parameters of the function "add".

2.
```
add ( i+1 , i=j+2 )
```

   This example passes the value of the expressions j+2 and i+1. The assignment in the expression can have unpredictable results.

Actual Parameters

Once an expression in a function call is evaluated, its value, called the "actual parameter", is assigned to the formal parameter that has the same place in the parameter list as the expression has in the expression list. For example, if the function "add" is declared as:

```
add (a, b)
int a, b;
{
        return ( a+b );
}
```

then the function call:

```
add(1, 2);
```

assigns 1 to the formal parameter "a" and 2 to the parameter "b". The number of expressions given in the expression list must match the number of formal parameters. The compiler does not check this number and supplying too many or too few causes an error during program execution.

There are few restrictions on the type, size, and number of actual parameters that can be passed. Actual parameters can be structures, unions, and pointers. Although entire arrays and functions cannot be passed, pointers to these items can. The following sections describe how a variety of variables are passed.

Fundamental Types

Actual parameters with fundamental type are passed by value. A copy of the actual parameter is assigned to the corresponding formal parameter and the function may use this copy without affecting the variables from which it was originally derived.

If the actual parameter has **char** type or is a bit field in structure, the value is converted to an **int** type before being passed to the function. The function, however, treats the value as the smaller type.

If the actual parameter is a **float** type, it is converted to **double** type and is treated as **double** type value by the function.

Arrays

An actual parameter cannot be an entire array. However, it can be a pointer to an array. Pointers to arrays are passed by address. This means a pointer to the actual array is assigned to the corresponding formal parameter. The formal parameter can then be used to access the actual contents of the array.

For example, in the following program fragment, the formal parameter str in the function strfind is assigned the address of the array "name" given in the function call in the **main** function.

```
main ()
{
    int x;
    char name[20];

    x = strfind(name, 'a');
}

strfind ( str, c )
char str[ ], c;
{
    int i;

    for (i=0; str[i] != 0 ; i++)
        if (str[i] == c)
            return (i);
    return(-1);
}
```

No copy of the array "name" is made. Instead, str receives the address of name, and, for all practical purposes, becomes name since all subsequent access to the elements of str yields direct access to the elements of name. Care must be taken when making assignments to formal parameters that correspond to arrays since changing an element in the formal parameter changes the original element as well.

In the above example, str is declared as an array with no explicit number of elements. This allows arrays of all sizes to be passed as actual parameters to str. Even if an explicit number of

elements were given, the compiler still allocates only enough space for the address of the array; it never allocates space for the entire array.

Although arrays are passed through pointers, an element of an array is always passed by value. This means a copy of the element is assigned to the corresponding formal parameter and subsequent access to that parameter does not affect the original element. For example, the function call:

```
strfind( name, name[3] );
```

assigns the value of the element name[3] to the second parameter of strfind, but assigns the address of the array "name" to the first parameter. If the element of an array is itself an array, it is passed through a pointer.

Structures and Unions

Structures and unions are passed by value. This means a copy of the entire contents of a structure or union is assigned to the corresponding formal parameter. The function can use this copy in expressions and assignments without affecting the contents of the structure or union from which it was originally taken.

The members of a structure or union are passed in the same way as variables with similar type, that is, integral members are passed by value and array members through pointers. Arrays of structures and arrays of unions are passed through pointers. However, elements of such arrays are passed by value.

Pointers

Pointers, which are passed by value, provide the only way to access a simple variable, structure, or union directly from a function. Since a pointer refers to the address of such a variable, the function can use this address to access the contents of the given variable. For example, in the following program fragment, the function swap exchanges the values of the variables x and y in the **main** function.

```
main ()
{
          int x, y;

          swap( &x, &y);
}

swap (a, b)
int *a, *b;
{
          int t;

          t = *a;
          *a = *b;
          *b = t;
}
```

In this case, the parameters a and b are declared as pointers to integer variables.

The function call:

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

assigns the address of x to a and the address of y to b. This means that references *a and *b in swap are, for all practical purposes, references to x and y in main . Thus, the subsequent assignments in swap change the contents of x and y directly. This same technique can be applied to structures and unions. The compiler automatically uses this technique when passing arrays.

If a formal parameter is declared to be a pointer, then any actual parameter to be assigned to it must be a valid address and have a pointer type. If the actual parameter does not have a pointer type, the compiler displays a warning message. If the address is not valid (for example, is not the address of a variable declared within the program), an error can occur during execution of the program.

Function Pointers

Although a function cannot be passed as an actual parameter, a pointer to a function can. This means the address of a function, like addresses of other items, can be passed to another function. In such cases, the address is assigned to the corresponding formal parameter which must be declared as a pointer to a function. The function containing the formal parameter can use the address to call the corresponding function.

For example, in the following program fragment, the **main** function selects a function for execution, then calls the function work, which actually calls the selected function.

```
main ()
{
   int lift(), step(), drop();
   int select, count;

       switch ( select ) {
           case 1:     work(count, lift);
                             break;

           case 2:     work(count, step);
                             break;
           case 3:     work(count, drop);
           default:
                             break;
       }
}

work ( n, func )
int n;
int (*func)();
{
        int i;

        for (i=0; i<n; i++)
           (*func)();
}
```

The formal parameter func in work is declared to be a pointer to a function. The parentheses around the parameter name are required since without them the declaration would define a function which returns a pointer to an integer.

The function call:

```
work (count, lift);
```

in **main** passes an integer variable and the address of the function lift. Only the name "lift" is given. An expression list and the parentheses that would normally enclose this list must not be given. This would cause the function lift to be called rather than have its address passed as an actual parameter. To use a function name in this way, a forward declaration for the function must be given before the name is used. In this case, the forward declaration is given at the beginning of the **main** function.

The function work calls the selected function by using the function call:

```
(*func)();
```

The call consists of the formal parameter func to which an expression list has been added. In this case, the expression list is empty. If the given function had formal parameters, appropriate actual parameters have to be supplied. If func is used by itself (that is, without an expression list), it refers to the address of the selected function and does not generate a function call.

Recursive Functions

All functions in a C program can be called recursively. This means, for example, that a function can call itself. The C compiler allows any number of recursive calls to a function. On each call, new storage is allocated for the formal parameters, and automatic and register variables so that their values in previous, unfinished calls will not be overwritten. Furthermore, previous parameters are inaccessible to all versions of the function except the version in which they were created.

Although the C compiler defines no limit on the number of times a function can be called recursively, there is a practical limit that can be imposed by the operating environment. Since each recursive call requires additional stack memory, too many recursive calls can cause a stack overflow on computers with fixed stacks.

Chapter 7. Preprocessor Directives

Contents

Introduction 7-3

Define Directive 7-4

Undefine Directive 7-7

Include Directive 7-8

If, Elif, Else, and Endif Directives 7-9

If defined and Elif defined Directives 7-12

Ifdef and Ifndef Directives 7-14

Line Control Directive 7-15

Introduction

This chapter describes the C language preprocessor directives and explains how to use these directives in C source programs. The preprocessor directives direct the C preprocessor to perform specific actions such as replace a given identifier with specified text or insert the contents of a file in the source program. The directives are typically used to make source programs easy to modify and easy to compile for different execution environments.

The compiler ordinarily invokes the preprocessor in its first pass, but the preprocessor can also be invoked separately to process text without compiling.

There are the following preprocessor directives:

#define
#undef
#include
#if
#else
#elif
#elif defined
#endif
#if defined
#ifdef
#ifndef
#line

Each directive starts with a number sign (#) as the first non-whitespace character in the line and may be followed by arguments or values. Directives can appear anywhere in a source file, but they apply only to the remainder of the source file in which they appear. The following sections describe each directive in detail.

Define Directive

Syntax

#define *identifier*[(*parameter-list*)] *text*

Description

The **define** directive replaces each subsequent occurrence of

identifier

with *text*. If a *parameter-list* is given, the **define** directive replaces each subsequent occurrence of

identifier(*argument-list*)

with a modified version of *text*. In this form, the values in the argument list replace the corresponding formal parameters in the *text.*

The *text* may be any text, such as keywords, constants, or complete statements. This text may also be empty, that is, have no characters. If the text is longer than can fit on a line, a backslash (\) can be used, escaping the newline character, to continue the text on a new line. The text must be separated from the identifier by at least one whitespace character.

The *parameter-list*, when given, must be enclosed in parentheses. No spaces between the identifier and this list are allowed. The parameter list may contain one or more formal parameter names. If several names are given, they must be unique within the list and must be separated by commas (,). The formal parameter names are used as placeholders in the *text* for values to be supplied later in an argument list. The parameter names can appear anywhere in the *text.* There is no limit to the number of formal parameters that can be used.

Once defined, an identifier can be used anywhere in the source program. When the program is processed, the identifier is replaced with its respective value. If an argument list is given with the identifier, the text is modified so that all formal

parameters in the string are replaced with the corresponding values given in the argument list. The number of arguments in the list must exactly match the number of formal parameters. If the text is empty, the identifier is removed from the source file unless it appears in an **if** directive. In this case, it is assumed to have the value 1.

A warning is issued if an identifier that takes arguments does not have an argument list. No warning is issued if the identifier appears in an **ifdef** directive.

An identifier may be redefined if its first definition is removed using the **undef** directive (described later in this chapter). If the definition is not removed, the preprocessor issues a warning. Redefining a directive to the same value causes no warning.

The **define** directive associates meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are called "manifest constants." Identifiers that represent statements or expressions are called "macros."

Examples

1. #define WIDTH 80

   This example defines the identifier WIDTH to be the integer constant 80.

2. #define MESSAGE "No space available."

   This example defines MESSAGE to be the string constant "No space available."

3. #define LENGTH WIDTH + 10

   This example defines LENGTH in terms of WIDTH and the integer constant 10. Each occurrence of LENGTH is replaced with WIDTH + 10 which is subsequently replaced with the expression 80+10.

4. #define BOOLEAN char

   This example defines BOOLEAN to be the keyword **char.** This means BOOLEAN can be used as if it were the keyword.

5. #define register

   This example defines **register ,** a keyword, to have no value. A program containing this definition removes each occurrence of **register** from the source file.

6. #define ADD(X,Y) X + Y

   This example defines a macro named ADD. This macro is replaced by the expression X + Y. For example, the occurrence ADD(1,2) is replaced with 1 + 2, and ADD(i,s[i]) is replaced with i + s[i].

7. #define MAX(x,y) x>y?x:y

   This example defines a macro named MAX. This macro is easier to read and understand than the corresponding expression, and therefore makes the source program easier to read.

Macros often look and act like C language function calls. In fact, carefully defined macros are frequently used in place of function calls to perform specific tasks. Unlike function calls, however, macros are replaced during preprocessing and do not appear in the executable program. This can make execution of the program faster since no call is actually performed, but it can make it harder to locate problems when debugging the program.

Undefine Directive

Syntax

```
#undef identifier
```

Description

The **undef** directive removes the current definition of *identifier*, causing subsequent occurrences of the identifier to be ignored by the preprocessor. Parameter lists are not allowed with the **undef**. The previous definition cannot be restored.

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. This is useful if a specific C language function in the source program has manifest constants that must take environment specific values that do not affect the rest of the program. The **undef** directive is also used with the **ifndef** directive (described later in this chapter) to conditionally control compilation of regions within the source program.

Example

```
#define WIDTH   80
#define ADD(X,Y)     X + Y
        .
        .
        .
#undef WIDTH
#undef ADD
```

In this example, the **undef** directive removes both a manifest constant and a macro. Note that only the identifier of the macro is given. An argument list is not allowed. The **undef** directive can also be used with identifiers that have no previous definition. This ensures that they are undefined.

Include Directive

Syntax

#include "*filespec*"
#include < *filespec* >

Description

The **include** directive causes the file named *filespec* to be inserted into the source program at the line containing the directive. The preprocessor processes the new text and passes it, along with the original source, to the compiler for compilation.

The *filespec* must name an existing file that contains appropriate text. If the file specification is enclosed in double quotation marks ("), the preprocessor searches for the file in the same directory as the source program file first, then searches directories specified in the compiler command line, and finally searches the standard directories. If the file specification is enclosed in angle brackets (<>), the preprocessor searches for the file in the specified or standard directories only. The syntax of the file specification and the locations of the standard directories depend on the operating system on which the program is compiled.

The **include** directive adds the definitions of useful constants and macros to a source program. The definitions can be collected in a single "include file" and then added to any source program that requires them by using an **include** directive.

The **include** directive can be nested. This means that the directive can appear in files named by other **include** directives. Nested **include** directives are processed after the file that contains them has been inserted into the source program. The nested directive causes the preprocessor to insert the contents of the given file into the source program just as if the directive were a part of the original source program. The new file can also contain **include** directives, and nesting can continue up to 10 levels.

Examples

1. #include <stdio.h>

2. #include "lib/defs.h"

The first example adds the contents of the file named stdio.h to the source program. The angle brackets cause the preprocessor to search the standard directories for stdio.h.

The second example adds the file specified by "lib/defs.h" to the source program. The double quotations marks cause the directory containing the current source file to be searched first.

If, Elif, Else, and Endif Directives

Syntax

#if *constant-expression*
    text
[#elif *constant-expression*
    text]
[#elif *constant-expression*
    text]
    .
    .
    .
[#else
    text]
#endif

Description

The **#if** directives, together with the **#elif** , **#else** , and **#endif** directives, allow you to control compilation of portions of a source file. The preprocessor selects one *text* block for further processing, based on the value of the *constant-expression* following

each directive. The selected text is processed by the preprocessor and passed to the compiler. The preprocessor ignores the remaining text blocks and does not pass them to the compiler.

The preprocessor selects a text block by evaluating the *constant-expression* following the **#if** directive. If the expression is true (nonzero), the text following the expression is selected. If the expression is false (zero), the preprocessor evaluates the *constant-expression* after the first **#elif** (else-if) directive. If that expression is true, the preprocessor selects the corresponding text. If not, it goes on to the next **#elif** directive and repeats the process until it finds a *constant-expression* with a true value.

The preprocessor selects the text after the **#else** clause if no *constant-expression* is true, or if there are no **#elif** directives. If the **#else** directive is omitted, and no *constant-expression* in the **#if** block is true, no text is selected.

Each *text* block is any text that has meaning to the compiler. It may occupy more than one line.

Each *constant-expression* is formed following the rules for constant-expressions discussed in Chapter 4, except that constant expressions in preprocessor directives may not contain **sizeof** expressions, type casts, or enumeration constants.

Each **#if** directive must be matched by an **#endif** directive to mark the end of the last text block. Any number of **#elif** directives (including zero) may appear between the **#if** and the **#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 be nested in the text portions of other **#if** directives. When nested, each **#else** , **#elif** , and **#endif** directive belongs to the closest preceding **#if** directive.

Examples

1.
```
#if DLEVEL > 5
        display( debugptr );
#endif
```

2.
```
#if DLEVEL > 5
        #define SIGNAL  1
        #if STACKUSE == 1
                        #define STACK   200
        #else
                        #define STACK   100
        #endif
#else
        #define SIGNAL  0
        #if STACKUSE == 1
                        #define STACK   100
        #else
                        #define STACK   50
        #endif
#endif
```

3.
```
#if DLEVEL == 0
        #define STACK 0
#elif DLEVEL == 1
        #define STACK 100
#elif DLEVEL > 5
        display( debugptr );
#else
        #define STACK 200
#endif
```

The above examples assume a previously defined manifest constant, DLEVEL . In the first example, the **#if** and **#endif** directives control compilation of statements used for debugging. The statement "display(debugptr)"is compiled only if the expression DLEVEL > 5 evaluates to a nonzero (true) value.

The second 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 second set is processed.

In the third example, **#elif** and **#else** directives are used to make one of four choices, based of the value of DLEVEL. The manifest constant STACK is set to 0, 100, or 200, depending on the definition of DLEVEL . If DLEVEL is not defined, " display(debugptr);" is compiled and STACK is not defined.

If defined and Elif defined Directives

Syntax

#if defined(*identifier*)
    text
#elif defined(*identifier*)
    text

Description

The **#if defined** and **#elif defined** directives have the same function as the **#if** and **#elif** directives, except that they test an identifier instead of a constant expression. If the given identifier is currently defined, the condition is considered to be true (nonzero). Otherwise, the condition is false (zero). An identifier defined as empty text is considered defined.

An **#if defined** directive may appear anywhere an **#if** directive appears, and, like the **#if** directive, must be matched with an **#endif** . Similarly, an **#elif defined** directive may appear anywhere an **#elif** directive appears. This means that a block beginning with **#if** or **#if defined** may contain both **#elif** and **#elif defined** directives, as well as an **#else** directive.

The *text* block to be preprocessed and compiled is selected exactly as outlined above for the **#if** directive.

Examples

1.
```
#if defined(M_86)
        #define SEG     67
        #define OFFSET  0
#endif
```

2.
```
#define REG1    register
#define REG2    register

#if defined(M_86)
        #define REG3
        #define REG4
        #define REG5
#else
        #define REG3    register
        #if defined(M_68000)
                #define REG4    register
                #define REG5    register
        #endif
#endif
```

In the first example, SEG and OFFSET are defined only if M_86 is previously defined.

The second example uses preprocessor directives to control the meaning of **register** declarations in a portable source file. The compiler assigns **register** storage to variables in the same order in which the **register** declarations appear in the source file. If you have more **register** declarations than the machine can accommodate, the compiler honors earlier declarations over later ones. This can result in a loss of efficiency if the variables declared later are more heavily used.

With the definitions listed above, you can give priority to your most important register declarations. REG1 and REG2 are defined as the **register** keyword to declare **register** storage for the two most important variables in the program.

For example, in the following fragment, b and c have higher priority than a or d.

```
foo(a)

REG3 a;

{
        REG1 b;
        REG2 c;
        REG4 d;
           .
           .
           .
}
```

When you define M__86, the preprocessor removes the REG3 identifier from the file by replacing it with empty text. This prevents "a" from receiving **register** storage at the expense of b and c. When M__68000 is defined, all four variables are declared to have **register** storage. When neither M__86 nor M__68000 is defined, a, b, and c, are declared with **register** storage.

Ifdef and Ifndef Directives

Syntax

```
#ifdef identifier
#ifndef identifier
```

Description

The **#ifdef** and **#ifndef** directives accomplish the same task as the **#if defined** directive and may be used anywhere **#if defined** may be used. These directives are provided only for compatibility with previous versions of the language. The **#if defined** directive is preferable for new code.

When the preprocessor encounters an **#ifdef** directive, it checks to see whether the *identifier* is currently defined. If so, the condition is true (nonzero). Otherwise, the condition is false (zero).

The **#ifndef** directive checks for exactly the opposite condition checked by **#ifdef** . If the identifier has not been defined (or its definition has been removed with **#undef**), the condition is true (nonzero). Otherwise, the condition is false (zero).

Line Control Directive

Syntax

#line *constant* ["*filename*"]

Description

The **line** directive causes the preprocessor to change the compiler's internally stored line number and filename to the given *constant* and *filename.* The compiler uses the internally stored line number and filename to refer to the location of any error encountered during compilation. The line number normally refers to the current input line; the filename refers to the current input file. The line number is incremented after each line is processed. Changing the line number and filename causes the compiler to ignore the previous values and to continue processing with the new values.

The *constant* can be any integer constant; the *filename* can be any combination of characters. It must be enclosed in double quotation marks ("). If no *filename* is given, the previous filename remains unchanged. The current line number and filename are always available using the predefined identifiers "__LINE__" and "__FILE__".

The **line** directive is used by program generators, such as **lex** and **yacc,** so that error messages refer to the original source file rather than the file created by the generator. The "__LINE__" and "__FILE__" identifiers insert error messages about the source file into the program text. See the second example below.

Example

1. #line 151 "copy.c"

2. #define ASSERT(cond)   if(!cond){printf("assertion\
   error line %d, file(%s)\n",__LINE__,__FILE____);}\
   else ;

In the first example, the internally stored line number is set to 151 and the file name is changed to copy.c .

In the second example, the macro ASSERT uses the predefined identifiers to create an error message about the source file. The text extends to several lines.

Appendixes

Contents

Appendix A. Differences **A-3**

Appendix B. C Compiler Messages and Limits **B-1**

Introduction **B-1**

Compiler Error Messages **B-1**
Warning Messages B-2
Program Error Messages B-8
Fatal Error Messages B-21

Compiler Limits **B-23**

Appendix A. Differences

This appendix outlines differences between IBM Personal Computer XENIX C Compiler and the description of the C language found in Appendix A of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, published in 1978 by Prentice-Hall, Englewood Cliffs, New Jersey. The differences are listed by the Kernighan/Ritchie section numbers.

2.2 Identifiers (including those used in preprocessor directives) are significant to 31 characters. External identifiers are also significant to 31 characters.

2.3 The identifiers **asm** and **entry** are no longer keywords. New keywords are **const, enum** and **void.** (Notice that **const** is not yet implemented but is reserved for future use.) The identifiers **near, far, pascal, fortran, huge** and **module** may be keywords, depending on whether the corresponding options are enabled when a program is compiled (see Chapter 2 in the *IBM Personal Computer XENIX Software Development Guide*) The **pascal, fortran** and **huge** options are not yet implemented.

2.4.1 Hexadecimal and octal constants are unsigned.

2.4.3 Hexadecimal bit patterns, consisting of a backslash (\), the letter 'x', and up to two hexadecimal digits, are permitted as character constants (for example, \x12).

There are two additional escape sequences. The sequence \v represents a vertical tab (VT), and the sequence \" represents the double quote character.

Character constants always have type **char,** with the result that they are sign-extended in type conversions.

2.6 The **short** type is always 16 bits long, the **long** type 32 bits. The size of an **int** is machine dependent. On 8086/8088 processors, an **int** is 16 bits long, while on 68000 machines, it is 32 bits.

4 The **char** type is signed, with the result that a **char** value is sign-extended in type conversions.

Two additional unsigned types are supported, **unsigned char** and **unsigned long**.

There is an additional fundamental type, the **enum** (enumeration) type. The **void** type is defined as the return type of functions that do not return a value.

6.5 The keyword **unsigned** may be applied as an adjective to any integer type (**char, int, short,** or **long**). When **unsigned** stands alone, it is taken to mean **unsigned int.**

6.6 The arithmetic conversions carried out by the IBM Personal Computer XENIX C Compiler are outlined in Chapter 4 of this reference. Although compatible with the Kernighan/Ritchie conversions, the IBM Personal Computer XENIX C Compiler conversions are spelled out in greater detail, including the specific path for each type of conversion.

7.2 In connection with the **sizeof** operator, a byte is defined as an 8-bit quantity.

7.14 A structure may be assigned to another structure of the same type.

8.2 The keywords **enum** and **void** are additional type specifiers. Additional acceptable combinations are **unsigned char, unsigned short, unsigned short int, unsigned long,** and **unsigned long int.**

8.5 Bit fields must be declared **unsigned**.

The names of structure members are not required to be distinct from structure tags or from the names of other variables.

There is no relationship between the members of two different structure types.

8.6 Unions may be initialized by giving a value for the first member of the union.

9.7 The *expression* of a **switch** statement has **enum** or integral type. Each of the **case** constant-expressions is cast to the type of the *expression*.

12 The pound sign (#) introducing the preprocessor directive may be preceded by any combination of whitespace characters, except for newline characters. There may also be whitespace (not including newlines) between the pound sign and the preprocessor keyword, and between the keyword and the opening parenthesis, "(," when arguments are present.

12.3 The new directive "**#if defined**(*identifier*)" is intended to supplant the "**#ifdef**" and "**#ifndef**" directives. Use of the latter directives is discouraged.

The new directive "**#elif**" (else-if) is designed for use in "**#if**" and "**#if defined**" blocks.

14.1 A structure may be assigned to another structure of the same type. Structures may be passed by value to functions and may be returned by functions.

In expressions involving "->," the expression before the arrow must have the same type (or be cast to the same type) as the structure to which the member on the right-hand side of the arrow belongs.

17 The listed anachronisms are not recognized.

6

Appendix B. C Compiler Messages and Limits

Introduction

This appendix lists the messages displayed by the **cc** command when errors are encountered during compilation of a program. It also lists the restrictions imposed by the compiler on the size and complexity of program source files and statements within source files.

Compiler Error Messages

The error messages produced by the C compiler fall into three categories: warnings, program errors, and fatal errors. Warnings alert you to problems that may cause errors during execution of the program, but do not prevent compilation of your program. Program errors identify problems that make successful compilation of your program impossible. Fatal errors identify problems that prevent **cc** from continuing execution. Whenever the compiler encounters program or fatal errors, it terminates operation before producing an object file.

The following sections explain the meaning of the compiler error messages, and provide clues on how to solve the problem indicated by these messages.

Warning Messages

The following is a complete list of compiler warnings messages. The number in square brackets ([ ]) at the end of each message gives the minimum warning level that must be set for the message to appear. You can set the warning level by using the **-W** option described earlier in this chapter.

warning: Address of frame variable taken, DS != SS [1]

Taking the address of a frame variable in a small model program with separate data and stack segments results in an incorrect address. The address does not refer to the correct segment.

warning: '*identifier*' : bad type (not integral) [1]

The given bitfield is converted to an *unsigned* integral type.

warning: '*identifier*' : bad type (not unsigned) [1]

The given bitfield is converted to an *unsigned* integral type.

warning: cast of int expression to far pointer [1]

A *far* pointer represents a full segmented address. Casting an integer value to a *far* pointer produces an address with a meaningless segment value.

warning: Constant too big [1]

Information is lost because a constant value is too large to be represented in the type to which it is assigned.

warning: conversion lost segment [1]

The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) results in the loss of the segment address.

warning: Data conversion [3]

Two data items had different types, causing the type of one item to be converted.

warning: '*operator*' : different types [1]
The values specified in the operation have different types.

warning: Float constant in a cross compilation [1]
Floating point constants are not portable because therepresentation of floating point values differs across machines.

warning: '*identifier*' : formals ignored [1]
Formal arguments appeared in a function declaration (for example, "extern int *f(a,b,c);"). The formal arguments are ignored.

warning: '*identifier*' : function as an argument [1]
A formal parameter to a function is declared to be a function, which is illegal. The formal parameter is converted to a function pointer.

warning: Function must return a value [2]
A function is expected to return a value unless it is declared as **void**.

warning: function *identifier* too large for post-optimizer [0]
The named function was not optimized because insufficient program space was available. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning: '*identifier*' : has bad class [1]
The specified storage class cannot be used in this context (for example, function parameters cannot be given *extern* class). The default storage class for that context is used in place of the illegal class.

warning: -S has precedence over -L [1]
You cannot create both a disassembled listing (-S) and an assembled listing (-L) with the same command. The -L option is ignored and a disassembled listing is created.

warning: Id truncated to *'identifier'* [1]
Only the first 31 characters of an identifier are significant.

warning: -C ignored (must also specify -P or -E or -EP) [1]
The -C option preserves comments in a preprocessed listing and takes effect only when you create such a listing with the -P, -E or -EP option.

warning: Ignoring unknown flag *option* [1]
The compiler does not recognize the given *option* and ignores it.

warning: Illegal null char [1]
The single quotes delimiting a character constant must contain one character. For example, the declaration "char a = ''" is illegal.

warning: '*operator*' : illegal pointer combination [1]
A pointer to a given type is forced to point to an object with a different type.

warning: '*operator*' : illegal with enums [1]
You may not use the given operator with **enum** values. The **enum** values are converted to **int** type.

warning: missing close paren after 'defined(id' [1]
The closing parenthesis is missing from an **#if defined** directive.

warning: Mixed near/far pointers [1]
A pointer is assigned to a pointer with a different size, resulting in the loss of a segment address from a **far** pointer or the addition of a segment address to a **near** pointer.

warning: Newline in string constant [1]
A newline character is not preceded by an escape character (\) in a string constant.

warning: '*identifier*' : no function return type [2]
A function declared to have **void** type returns a value.

warning: No return value [2]

A function declared to return a value does not do so.

warning: Not enough parameters [1]

The number of actual arguments specified with an identifier is less than the number of formal parameters given in the macro definition of the identifier.

warning: '&' on function/array, ignored [1]

The address of (&) operator is used incorrectly on a function or array.

warning: Only one of **-P/-E/-EP** allowed, **-P** selected [1]

Each of the **-P**, **-E** and **-EP** options produces a different kind of preprocessed listing; only one option can be used at a time.

warning: overflow in constant arithmetic [1]

The result of an operation exceeds 0x7FFFFFFF.

warning: overflow in constant multiplication [1]

The result of an operation exceeds 0x7FFFFFFF.

warning: '*identifier*' : overflows array bounds [1]

Too many initializers are present for the given array. The excess initializers are ignored.

warning: Pointer mismatch. [1]

Pointers to different types of variables are used interchangeably.

warning: Procedure too large, loop inversion optimization missed but continuing [0]

Some optimizations for a function are skipped because insufficient program space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning: Procedure too large, skipping branch sequence optimization and continuing [0]

Some optimizations are skipped because insufficient program space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning: Procedure too large, skipping cross jump optimization and continuing [0]

Some optimizations for a function are skipped because insufficient program space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning: Recoverable heap overflow in post optimizer - some optimizations may be missed [0]

Some optimizations are skipped because insufficient program space is available for optimization. To correct this program, reduce the size of the function by breaking it down into two or more smaller functions.

warning: '*identifier*' : redeclaration ignored [1]

The named formal parameter was previously defined.

warning: '*identifier*' : redefinition [1]

The given *identifier* is redefined.

warning: 'register' on '*identifier*' ignored [1]

Only integral and pointer type variables may be given **register** storage class.

warning: "-i" required on the command line, changing name segment or group requires separate i and d. Setting -i and continuing. [1]

The text segment or group of a small model program can be renamed (using -NT or -NGT) only if a separate text segment is created using the -i option.

warning: requires parameters [1]

Formal parameters are given in the macro definition of an identifier, but no argument list is given with the identifier.

warning: Storage class *class* on '*identifier*' changed to extern [1]

Items declared outside of functions must have **static** or **extern** storage class.

warning: String too big, leading chars truncated [1]

Strings may not exceed 512 bytes.

warning: Strong type mis-match [2]

Two different but compatible types are used: for example, a **typedef** type with a non-**typedef** type, or two different but equivalent **struct** or **union** types.

warning: Too many parameters [1]

The number of actual arguments specified with an identifier is greater than the number of formal parameters given in the macro definition of the identifier.

warning: Type following '*keyword*' is illegal, ignored [1]

An illegal combination occurs (for example, **unsigned float**.)

warning: *identifier* : undefined [1]

The given identifier is not defined. is not defined.

warning: '*identifier*' : unknown array size [1]

The size of the named array is not specified.

warning: '*identifier*' : unknown size [1]

The size of the named variable is not specified.

warning: unmatched close comment '*/' [1]

A comment was started (with '/*:') but was not closed (with '*/').

warning: '*identifier*' : void type changed to int [1]

Only functions may be declared to have **void** type.

Program Error Messages

The following is a complete list of program error messages. After printing a program error message, the compiler typically continues to look for more errors, but will not create an object file.

'+': 2 pointers

Two pointers may not be added.

'***identifier***': aggregate inits require curly braces

An initializer for an aggregate type has a syntax error.

Array of functions

Arrays of functions are not allowed.

auto allocation exceeds 32KB

The space allocated for the local variables of a function exceeds the limit of 32KB.

'***identifier***' : automatic struct/arrays

Structures, arrays, and unions with **auto** storage class cannot be initialized.

Bad call

The expression before the parentheses in a function call does not evaluate to a function pointer. For example,

```
int *p;
   .
   .
   .
(*p)();
```

'***class***': bad class

The given storage *class* cannot be used in this context.

operator : bad left operand

The left-hand operand of the given *operator* is an illegal value.

Bad octal number '*n*'

The character *n* is not a valid octal digit.

operator : bad right operand

The right-hand operand of the given *operator* is an illegal value.

'***identifier***' : base type with near/far not allowed

Declarations of structure and union members may not use the **near** and **far** keywords to override the addressing convention for a member.

can't cast objects as 'far'

The **near** and **far** keywords may not be used in type casts. For example, "(int far)foo" is illegal.

can't cast objects as 'near'

The **near** and **far** keywords may not be used in type casts. For example, "(int near)foo" is illegal.

Case expression not constant

Case expressions must be integral constants.

Case expression not integral

Case expressions must be integral constants.

Case value '*n*' already used

The case value *n* has already been used in this **switch** statement.

cast of 'void' term to non-void

The **void** type may not be cast to any other type.

cast to array type is illegal

An object cannot be cast to an array type.

cast to function returning . . . is illegal

An object cannot be cast to a function type.

Compiler error (assertion): file *filename*, line *n* source=*filename*

The compiler consistency check failed. Try rearranging your code. In this message, the first *filename* identifies the compiler file producing the

error; the line number *n* refers to that file. The second *filename* gives the name of the source file being compiled.

Compiler error (code generation)
The compiler could not generate code for this expression. Try rearranging the expression.

Compiler error (internal): . . . .
The compiler consistency check failed. Try rearranging your code.

Compiler limit: macro's actual parameter is too big
Arguments to preprocessor macros may not exceed 256 bytes.

Compiler limit: struct/union nesting
Nesting of structure and union definitions may not exceed 5 levels.

Compiler limit: Too many actual parameters for macro
A macro definition may not take more than 8 actual arguments.

compiler limitation: Initializers too deeply nested
The compiler limit on nesting of initializers has been exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized. To correct this problem, simplify the data type being initialized to reduce the levels or nesting, or assign initial values in separate statements after the declaration.

Constant expression is not integral
The context requires an integral constant expression.

#define syntax
A ***#define*** directive has a syntax error.

'***identifier***': definition too big
Macro definitions may not exceed 256 bytes.

'operator': different aggregate types

Pointers to different structure or union types are not allowed with the given *operator*.

Divide by 0

The second operand in a division (/) operation evaluates to zero (0).

'identifier': enum/struct/union type redefinition

The given *identifier* has already been used for an enumeration, structure, or union tag in the same scope.

expected '(' to follow 'identifier'

The context requires parentheses after the function *identifier*.

Expected constant expression

The context requires a constant expression.

expected 'defined(id)'

An **#if defined** directive has a syntax error.

Expected exponent value, not '*n*'

The exponent of a floating point constant is not a valid number.

Expected preprocessor command, found '*c*'.

The character following a number sign (#) is not the first letter of a preprocessor directive.

'identifier': field is an array/ptr

Bitfield members must have *unsigned* integral type.

'*identifier*': field type too small for number of bits

The number of bits specified in the bitfield declaration exceeds the number of bits in an **unsigned** integer of the given size.

'*identifier*': fields only in structs

Only structure types may contain bitfields.

Function returns array

A function may not return an array. (It may return a pointer to an array.)

Function returns function

A function may not return a function. (It may return a pointer to a function.)

'***identifier***': Functions are illegal members

A function cannot be a member of a structure; use a pointer to a function instead.

'***string***': ignored

The given text appeared out of context and was ignored.

Illegal allocation of segment > 64KB

The space allocated for a single data item exceeds the limit of one segment (64KB).

Illegal break

A **break** statement is legal only when it appears within a **do, for, while**, or **switch** statement.

Illegal case

The **case** keyword may only appear within a **switch** statement.

illegal cast

A type used in a cast operation is not a legal type.

Illegal continue

A **continue** statement is legal only when it appears within a **do, for**, or **while** statement.

Illegal default

The **default** keyword may only appear within a **switch** statement.

Illegal escape sequence

The character(s) after the escape character (\) do not form a valid escape sequence.

Illegal expression

An expression is illegal because of a previous error. (The previous error may not have produced an error message.)

'***operator***' : illegal for struct/union

Structure and union type values are not allowed with the given *operator*.

Illegal index, indirection not allowed

A subscript was applied to an expression that does not evaluate to a pointer.

Illegal indirection:

The indirection operator ('*') was applied to a non-pointer value.

Illegal initialization.

An initialization is illegal because of a previous error. (The previous error may not have produced an error message).

'***operator***': illegal pointer combination

Pointers that point to different types cannot be used with the given *operator*

Illegal pointer subtraction.

Only pointers that point to the same type may be subtracted.

#include expected a file name

An **#include** directive lacks the mandatory filename specification.

'***identifier***': init of a function

Functions may not be initialized.

'***identifier***' is an undefined struct/union

The structure or union type of the given *identifier* is not defined.

keyword 'enum' illegal

The **enum** keyword appears in a structure or union declaration, or an **enum** type definition is not formed correctly.

Label '*identifier*' was undefined

The function does not contain a statement labeled with the given *identifier*.

left of '->*identifier*' must have a struct/union type

The expression before the member selection operator '->' does not point to a structure or union type.

left of '.*identifier*' must have a struct/union type

The expression before the member selection operator '.' does not have a structure or union type.

left of '->' specifies undefined struct/union '*identifier*'

The expression before the member selection operator '->' points to a structure or union type that is not defined.

left of '.' specifies undefined struct/union '***identifier***'

The expression before the member selection operator '.' has a structure or union type that is not defined.

operator: Left operand must be lval.

The left operand of the given *operator* must be an lvalue.

#line expected a line number

A **#line** directive lacks the mandatory line number specification.

'***identifier***': member of enum redefinition

The given *identifier* has already been used for an enumeration constant, either within the same enumeration type or within another enumeration type in the same scope.

Missing '>'

The closing angle bracket ('>') is missing from an **#include** directive.

Missing name following '<'

An **#include** directive lacks the mandatory filename specification.

missing open paren after keyword 'defined'

Parentheses must surround the identifier to be checked in an **#if** defined directive.

'***identifier***' : Missing subscript

To reference an element of an array you must use a subscript (for example, "A[6]").

Mod by 0

The second operand in a remainder (%) operation evaluates to zero (0).

More than one default

A switch statement contains too many default labels (only one is allowed).

'*operator*' needs lvalue.

The given *operator* must have an lvalue operand.

negative subscript

A value defining an array size was negative.

Newline in constant

A newline character in a character or string constant must be preceded by the backslash escape character (\).

No closing single quote

A newline character in a character constant must be preceded by the backslash escape character (\).

No struct definition

A structure or union type is used in a declaration without being defined.

Non-address expression

An attempt was made to initialize an item that is not a lvalue. For example, the declaration " int i, j = 1; " in the following example is illegal.

```
int i, j = i;
main()
     .
     .
     .
}
```

The declaration occurs outside of all functions, so it cannot be determined until link time (too late for initialization) whether *i* is a reference to a global variable defined and initialized elsewhere, or a definition of a global variable (with a default initial value of 0) .

Non-constant offset

An initializer uses a non-constant offset. For example, the declaration " int i, j, *p = &i + j; " in the following example is illegal.

```
int i, j, *p = &i + j;
main()
{
     .
     .
     .
}
```

The declaration occurs outside of all functions, so it cannot be determined until link time (too late for initialization) whether *i* and *j* are references to global variables defined and initialized elsewhere, or definitions of global variables (with default initial values of 0).

Non-integer switch expression

Switch expressions must be integral.

Non-integral index

Only integral expressions are allowed in array subscripts.

'*identifier*': not a function

The given *identifier* was not declared as a function but an attempt was made to use it as a function. For example,

```
int i;
   .
   .
   .
i();
```

'*identifier*': not a label

The *identifier* specified in a goto statement does not correspond to a statement label.

'identifier': not struct/union member

The given *identifier* is used in a context that requires a structure or union member.

'&' on bit field ignored

Bitfields cannot have their address taken.

'&' on constant

Only variables and functions can have their address taken.

'&' on register variable

Register variables cannot have their address taken.

parameter has type void

Only functions have **void** type, and formal parameters may not be functions.

pointer + non-integer

Only integral values may be added to pointers.

'*operator*': pointer on left. Needs integral right.

The left operand of the given **operator** is a pointer; the right operand must be an integral value.

'+' 2 pointers

Two pointers may not be added.

Preprocessor command must start as first non-white

Non-whitespace characters appear before the number sign (#) of a preprocessor directive on the same line.

'identifier': redefinition

The given *identifier* was defined more than once in the same scope.

'.' requires struct/union name

The expression before the member selection operator '.' is not the name of a structure or union.

'->' requires struct/union pointer

The expression before the member selection operator '->' is not a pointer to a structure or union.

'-': right operand pointer

If the left-hand operand in a subtraction (-) operation is not a pointer, the right-hand operand is not permitted to be a pointer.

***Static procedure 'identifier'* not found**

A forward reference was made to a missing static procedure.

Structure/Union comparison illegal

You cannot compare a structure type to a union type. (You can, however, compare individual members of structure and unions).

Subscript on non-array

A subscript was used on a variable that is not an array.

syntax error

This statement or the preceding statement is not formed correctly.

'*n*': too big for char

The number *n* is too large to be represented as a character.

Too many chars in constant

A character constant is limited to a single character. (Multi-character character constants are not supported).

Too many initializers

The number of initializers exceeds the number of objects to be initialized.

Typedef specifies different enum

Two enumeration types defined with **typedef** are used to declare an item, but the enumeration types are different.

Typedef specifies different struct

Two structure types defined with **typedef** are used to declare an item, but the structure types are different.

Typedef specifies different union

Two union types defined with typedef are used to declare an item, but the union types are different.

'typedefs' both define indirection

Two **typedef** types are used to declare an item and both **typedef** types have indirection. For example, the declaration of *p* in the following example is illegal.

```
typedef int *P_INT
;
typedef short *P_SHORT
;
 P_SHORT P_INT p
;  /* this declaration is illegal */
```

'***identifier***': undefined

The given *identifier* is not defined.

'*c*': unexpected in formal list

The character *c* is misused in a macro definition's list of formal parameters.

'*c*': unexpected in macro definition

The character *c* is misused in a macro definition.

unknown character '0x*n*'

The given hexadecimal number does not correspond to to a character in the C character set.

'***identifier***': unknown size

A member of a structure or union has an undefined size.

'void' illegal with all types

The **void** type cannot be used in operations with other types.

'***expression***' was the use of the struct/union

An undefined structure or union type variable is used in the given *expression*.

Fatal Error Messages

The following is a complete list of fatal error messages. After printing a fatal error message, the compiler terminates processing and returns control to the system.

fatal : Bad flag = *option*

The given *option* is illegal or inconsistent with another option appearing on the same line.

fatal : Bad parenthesis nesting

The parentheses in a preprocessor directive are not matched.

fatal : Bad preprocessor command *'string'*

The characters following the number sign (#) do not form a preprocessor directive.

fatal: Cannot open *'filename'*

The compiler ran out of disk space, or the disk is protected against writing.

fatal : Compiler limit : Macro expansion too big

The expansion of a macro exceeds the space available for it.

fatal : Compiler limit : possibly a recursively defined macro

The expansion of a macro exceeds the space available for it. Check to see whether the macro is recursively defined.

fatal DGROUP data allocation exceeds 64KB

Long model allocation of variables to the default segment exceeds 64KB.

fatal #if[n]def expected an identifier

You must specify an identifier with the **#ifdef** and **#ifndef** directives.

fatal : expected '#endif'

An **#if**, **#ifdef**, **#ifndef**, or**#if defined** directive was not terminated with an **#endif** directive.

fatal : only one memory model allowed

Conflicting memory model options appear on the command line.

fatal : Parser stack overflow, please simplify your program

Your program is cannot be processed because the space required to parse the program exceeds a compiler limit. To solve this problem, try to simplify your program.

fatal : Too many include files

Nesting of **#include** directives exceeds the limit of 10 levels.

fatal unexpected '#elif'

The **#elif** directive is legal only when it appears within an **#if, #if defined, #ifdef,** or **#ifndef directive.**

fatal unexpected '#else'

The **#else** directive is legal only when it appears within an **#if, #if defined, #ifdef,** or **#ifndef** directive.

fatal unexpected '#endif'

An **#endif** directive appears without a matching **#if, #if defined, #ifdef,** or **#ifndef directive.**

fatal : Unexpected **EOF**

The end of a file was encountered. This message appears when you have insufficient space on the default disk drive for the compiler to create the temporary files it needs. The space required is approximately three times the size of the source file.

fatal : Unknown configuration string *'string'*

The configuration string given with the **-M** option contains an unrecognized character.

fatal : Unknown model type

The configuration string given with the **-M** option contains an unrecognized character.

Compiler Limits

The following list summarizes the limits imposed by the C compiler. If your program exceeds any of these limits, an error message will inform you of the problem.

1. **Disk Space**

Minimum disk space for compilation	3 times source file size

2. **Declarations**

Maximum number of dimensions in an array	5 dimensions
Maximum level of nesting for structure/union definitions	5 levels
Maximum level of indirection	5 levels
Maximum level of nesting for aggregate initializers	10-15 levels

This depends on the combination of aggregate types; higher levels of nesting are possible with array initialization than with structure and union initialization.

3. **Constants**

Maximum length of a string, including the terminating null character (\0)	512 bytes

4. **Identifiers**

Maximum length of an identifier	31 characters

5. **Preprocessor Directives**

Maximum size of a macro definition	512 bytes

Maximum number of actual arguments to a macro definition	8 arguments
Maximum length of an actual preprocessor argument	256 bytes
Maximum level of nesting for **#if**, **#ifdef**, **#ifndef**, and **#if defined** directives	32 levels
Maximum level of nesting for include files	10 levels

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

Index

Special Characters

& operator 4-37
| operator 4-37
^ operator 4-37
(&&)operator 4-38
(~) bitwise complement 4-28
(<<) operator 4-35
(+) addition 4-32
(||)operator 4-38
(&) operator 4-29
(!) logical not 4-28
(*) indirection 4-29
(*) multiplication 4-31
(-) arithmetic negation 4-28
(-) subtraction 4-33
(/) division 4-31
(%) remainder 4-31
(>>) operator 4-35
#if defined directive 7-12
#ifdef directive 7-14
#ifndef directive 7-14

A

addition (+) 4-32
additive operators 4-32
address of (&) operator 4-29
address of operators 4-29
angle brackets (< >) 7-8
argument list 7-5
arithmetic negation (-) 4-28
arithmetic operators 1-8
array declarations 3-28
array modifier 3-15
asm 1-15
assignment conversions 4-18
assignment operators 4-42
assignments 4-5
auto 1-15
automatic class 3-4
    auto 3-4

B

bit field 3-23
bitwise AND operator 4-37
bitwise complement (~) 4-28
bitwise exclusive OR
  operator 4-37
bitwise operators 4-37
bitwise OR operator,
  exclusive 4-37
block level nesting 2-10
break 1-15

C

case 1-15
cast, type 4-25
char 1-15
character

digits 1-5
escape sequence 1-7
letters 1-5
non-graphic 1-7
punctuation 1-6
special 1-6
whitespace 1-5
character constants 1-12
character set 1-5
class,extern 3-6
class,register 3-5
class,static 3-5
comments 1-15
Compilation, controlling with preprocessor directives 7-12, 7-14
compiler limits B-23
compiler requirements B-23
complement,bitwise 4-28
complex declarators 3-17
compound assignment 4-46
constant expressions 4-16
constant types
char 1-10
long 1-10
short int 1-10
unsigned 1-10
constants 4-6
character 1-12
decimal 1-9
floating-point 1-11
hexadecimal 1-9
integer 1-9
manifest 7-5
octal 1-9
string 1-12
continue 1-15
conversions
assignment 4-18
from floating point types 4-22
from pointer types 4-24
from signed integer types 4-18
from structure and union types 4-24
from unsigned integer types 4-20
from void types 4-24
function call 4-26
operator 4-25
type 4-17
type cast 4-25

D

declarations 3-3
array 3-28
declarations,pointer 3-33
function 3-35
simple variable 3-27
structure 3-30
typedef 3-36
union 3-24, 3-31
variable 3-26
visibility and scope 3-44
declarators
array modifier 3-15
declarators,complex declarators 3-17
function modifier 3-17
pointer modifier 3-14
default 1-15
define 7-3
definitions 2-3
directives 2-3, 2-9
division (/) 4-31
do 1-15
double 1-15
double quotation marks (") 7-8

E

elif 7-3
elif defined 7-3
elif directives 7-9
else 1-15, 7-3
else directives 7-9
endif 7-3
endif directives 7-9
entry 1-15
enum 1-15
enumeration declarations 3-21
enumeration types 3-9
error messages B-1
escape sequences 1-7
expressions 4-14
    constant 4-16
    parenthesized 4-16
    type cast 4-15
    with operators 4-15
extern 1-15
external class 3-6
external functions 3-47
external variables 3-45

F

float 1-15
floating-point constants 1-11
for 1-15
fortran 1-15
function 2-7
    calling 2-7
    main program 2-7
function call conversions 4-26
function calls 4-9
function declarations 3-35
function definition 2-4
function modifier 3-17
functions external 3-47
functions global 3-47
functions,static 3-47
fundamental types 3-8

G

global functions 3-47
global level nesting 2-10
global variables 3-45
goto 1-15

I

identifier 4-6
    array 4-7
    enumeration 4-6
    floating point 4-6
    function 4-7
    integral 4-6
    pointer 4-7
    predefined 1-15
    structure 4-7
    union 4-7
if 1-15, 7-3
if defined 7-3
If defined directive 7-12
if directives 7-9
ifdef 7-3
Ifdef directive 7-14
ifndef 7-3
Ifndef directive 7-14
include 7-3
include directive 7-8
indirection (*) 4-29

indirection and address of operators 4-29
initialization
- aggregate type 3-40
- fundamental type 3-38
- pointer type 3-39
- string type 3-43

int 1-15
integer and pointer combinations 4-33
integer, unsigned (type conversion) 4-20

K

keywords 1-15

L

left shift (<<) operator 4-35
letters and digits 1-5
limits, compiler B-23
line 7-3
line control directive 7-15
logical AND (&&) 4-38
logical not (!) 4-28
logical operators 1-8, 4-38
logical OR (| |) 4-39
long 1-15

M

macros 7-5
manifest constants 7-5
member selection expressions 4-13
messages B-1
multiplication (*) 4-31
multiplicative operators 4-31

N

nesting,block 2-10
nesting,global 2-10
nontrivial program 2-4
notational conventions 1-3
null statement 5-17
number sign(#) 7-3

O

operands
- constant 4-6
- expressions 4-14
- function calls 4-9
- member selection expressions 4-13
- operands,identifiers 4-6
- strings 4-8
- subscript expressions 4-10

operator
- additive 4-32
- address of 4-29
- address of (&) 4-29

arithmetic 1-8
assignment 1-8, 4-42
binary 4-27
bitwise AND (&) 4-37
bitwise exclusive OR 4-37
conditional 4-41
indirection (*) 4-29
left shift (<<) 4-35
logical 1-8, 4-38
logical AND (&&) 4-38
logical OR (| |) 4-39
negation 4-28
relational 4-36
right shift (>>) 4-35
sequential evaluation 4-40
shift 4-35
sizeof 4-30
ternary 4-27
unary 4-27
operator conversions 4-25
operators 4-27
overflow 4-35

P

parameter-list 7-4
parenthesized expressions 4-16
pointer and integer combinations 4-33
pointer declarations 3-33
pointer modifier 3-14
pointer, type conversion 4-24
precedence 4-47
preprocessor directives
define directive 7-4
elif 7-9
else 7-9
endif 7-9
if 7-9
include 7-8
line control 7-15
undefine 7-7
Preprocessor directives, #if defined 7-12
Preprocessor directives, #ifdef 7-14
Preprocessor directives, #ifndef 7-14
program execution 2-7

Q

quotation marks, double 7-8

R

register 1-15
register class 3-5
relational operators 4-36
remainder (%) 4-31
return 1-15
right shift (>>) operator 4-35

S

scope 2-10
sequential evaluation operator 4-40
shift operators 4-35
short 1-15
side effects 4-51
simple assignment 4-45

INDEX

simple variable
declarations 3-27
sizeof 1-15
sizeof keyword 4-30
sizeof operator 4-30
source Files 2-5
source program 2-3
statements, null
statement 5-17
static 1-15
static class 3-5
static functions 3-47
static variables 3-46
storage class specifiers
automatic 3-4
external 3-4
register 3-4
static 3-4
string constant 1-12
strings 4-8
struct 1-15
structure declarations 3-22,
3-30
structure types 3-10
subscript expressions 4-10
subtraction (-) 4-33
switch 1-15

T

the conditional operator 4-41
Tokens 1-17
type cast expressions 4-15
type conversions
assignment 4-18
floating point types 4-22
integer, unsigned 4-20
pointers 4-24
signed integer 4-18
structure and union
types 4-24
unsigned integer 4-20
void 4-24
type declarations 3-20
type
declarations,enumeration 3-21
type
declarations,structure 3-22
type declarations,union 3-24
type names 3-48
type specifiers 3-7
char 3-7
double 3-7
enumeration 3-7
float 3-7
long int 3-7
short int 3-7
structure 3-7
type
specifiers,enumeration 3-9
type
specifiers,fundamental 3-8
type
specifiers,structure 3-10
type specifiers,union 3-12
union 3-7
unsigned char 3-7
unsigned long int 3-7
unsigned short int 3-7
void 3-7
typedef 1-15
typedef declarations 3-36

U

unary increment and
decrement 4-44
undef 7-3
undef directive 7-7

undefine directive 7-7
union 1-15
union declarations 3-24, 3-31
union types 3-12
unsigned 1-15
unsigned integer (type conversion) 4-20

V

variable declarations 3-26
variable definitions 2-4
variable initialization 3-38
variables
  external 3-45
  global 3-45
  static 3-46
visibility 2-10
  nesting 2-10
visibility and scope 3-44
void 1-15
void, type conversion 4-24

W

while 1-15

INDEX