SC09-1128-01

C Language Manual

IBM

SC09-1128-01

C Language Manual

Second Edition (October 1987)

This edition applies to the IBM C for System/370 Program Offering (Products 5713-AAG and 5713-AAH).

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to

IBM Canada Ltd.
Information Development,
Department 849,
1150 Eglinton Ave. East
North York, Ontario, Canada. M3C 1H7

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

C Lang.

Contents

Chapter 1: Introduction

Features of C for System/370 ..... 1 - 2
The C Language Standard ..... 1 - 3
IBM Extensions to the Proposed ANSI Standard ..... 1 - 3
C Program Structure ..... 1 - 5
Functions and Data Objects ..... 1 - 5
C Source Files ..... 1 - 5
Preprocessor Capabilities ..... 1 - 5
Compilations ..... 1 - 6

Chapter 2: Elements of the C Language

The Character Set for C ..... 2 - 1
Source Lines ..... 2 - 2
Whitespace and Comments ..... 2 - 2
Identifiers ..... 2 - 3
Keywords ..... 2 - 4
Constants ..... 2 - 4
Integer Constants ..... 2 - 4
Floating Constants ..... 2 - 5
Character Constants ..... 2 - 6
String Constants ..... 2 - 7
Delimiters, Separators, and Operators ..... 2 - 9

Chapter 3: Identifiers

Naming Things in C ..... 3 - 1
Declaring Identifiers ..... 3 - 3
Name Spaces ..... 3 - 6
Scope of Names ..... 3 - 6
Visibility ..... 3 - 7
Linkage ..... 3 - 9

Chapter 4: Declarations

Components of a Declaration ..... 4 - 1
Storage Class ..... 4 - 2
Initial Declarations ..... 4 - 3
Redeclarations ..... 4 - 5
Base Type ..... 4 - 7
The Void Type ..... 4 - 9
The Integer Types ..... 4 - 9
char type ..... 4 - 9
signed char type ..... 4 - 10
unsigned char type ..... 4 - 10
short type ..... 4 - 10
unsigned short type ..... 4 - 10
int type ..... 4 - 10
unsigned int type ..... 4 - 10
long type ..... 4 - 11

unsigned long type ..... 4 - 11
The Floating Types ..... 4 - 11
float type ..... 4 - 11
double type ..... 4 - 11
long double type ..... 4 - 12
The Composite Types ..... 4 - 12
Enumeration Types ..... 4 - 12
Struct Types ..... 4 - 13
Bitfields ..... 4 - 14
Union Types ..... 4 - 15
Type Qualifiers ..... 4 - 16
The Type Qualifier const ..... 4 - 16
The Type Qualifier volatile ..... 4 - 17
Declarators ..... 4 - 17
Pointer Type Attribute ..... 4 - 18
Array Type Attribute ..... 4 - 19
Function Type Attribute ..... 4 - 20
Function Prototype Arguments ..... 4 - 21
OS Calling Sequence ..... 4 - 21
Definitions ..... 4 - 22
Function Definition ..... 4 - 22
Data Object Definition ..... 4 - 22
Data Initializers ..... 4 - 23
Type Definitions ..... 4 - 25
Type Names ..... 4 - 25
Classification of Types ..... 4 - 26

Chapter 5: Expressions
Grouping and Precedence ..... 5 - 1
Regrouping ..... 5 - 3
Side Effects and Order of Evaluation ..... 5 - 4
Types and Classes of Expressions ..... 5 - 4
Types of Expressions ..... 5 - 5
Classes of Expressions ..... 5 - 5
Arithmetic Conversions ..... 5 - 7
Widening Conversions ..... 5 - 7
Unsignedness Preserving Rules ..... 5 - 9
Narrowing Conversions ..... 5 - 9
Unsigned Conversions ..... 5 - 10
Pointer Conversions ..... 5 - 10
Converting Integers to Pointers ..... 5 - 10
Converting Function Pointers ..... 5 - 11
Converting Data Object Pointers ..... 5 - 11
Converting Incomplete Pointers ..... 5 - 12
Converting Pointers to Integers ..... 5 - 12
Pointer Arithmetic ..... 5 - 12
Comparing Types ..... 5 - 13
Same Types ..... 5 - 13
Assignment Compatibility ..... 5 - 14
The C Operators ..... 5 - 15
Addressing Operators ..... 5 - 15
Subscript Operator ..... 5 - 15
Point at Member Operator ..... 5 - 16
Select Member Operator ..... 5 - 16

Function Calls ..... 5 - 16
OS Calling Sequence ..... 5 - 19
Unary Operators ..... 5 - 19
Logical NOT Operator ..... 5 - 19
Bitwise NOT Operator ..... 5 - 19
Preincrement Operator ..... 5 - 19
Predecrement Operator ..... 5 - 19
Postincrement Operator ..... 5 - 20
Postdecrement Operator ..... 5 - 20
Plus Operator ..... 5 - 20
Minus Operator ..... 5 - 20
Indirect On Operator ..... 5 - 20
Address Of Operator ..... 5 - 21
Size Of Operator ..... 5 - 21
Type Cast Operators ..... 5 - 21
Multiplicative Operators ..... 5 - 22
Multiply Operator ..... 5 - 22
Divide Operator ..... 5 - 22
Remainder Operator ..... 5 - 22
Additive Operators ..... 5 - 22
Add Operator ..... 5 - 22
Subtract Operator ..... 5 - 23
Bitwise Shift Operators ..... 5 - 23
Left Shift Operator ..... 5 - 23
Right Shift Operator ..... 5 - 23
Relational and Equality Operators ..... 5 - 24
Less Than Operator ..... 5 - 24
Less Than Or Equal To Operator ..... 5 - 24
Greater Than Operator ..... 5 - 24
Greater Than Or Equal To Operator ..... 5 - 24
Equal To Operator ..... 5 - 25
Not Equal To Operator ..... 5 - 25
Bitwise Binary Operators ..... 5 - 25
Bitwise AND Operator ..... 5 - 25
Bitwise Exclusive OR Operator ..... 5 - 26
Bitwise Inclusive OR Operator ..... 5 - 26
Logical and Conditional Operators ..... 5 - 26
Logical AND Operator ..... 5 - 26
Logical OR Operator ..... 5 - 26
Conditional Operator ..... 5 - 27
Assignment Operators ..... 5 - 27
Gets Operator ..... 5 - 27
Gets Multiplied Operator ..... 5 - 27
Gets Divided Operator ..... 5 - 28
Gets Remainder Operator ..... 5 - 28
Gets Added Operator ..... 5 - 28
Gets Subtracted Operator ..... 5 - 28
Gets Left Shifted Operator ..... 5 - 28
Gets Right Shifted Operator ..... 5 - 28
Gets AND Operator ..... 5 - 28
Gets Exclusive OR Operator ..... 5 - 28
Gets Inclusive OR Operator ..... 5 - 28
Comma Operator ..... 5 - 29
Constant Expressions ..... 5 - 29

Contents

Chapter 6: Statements

Test and Void Expressions ..... 6 - 1
Labels ..... 6 - 2
Plain Label ..... 6 - 2
Case Label ..... 6 - 2
Default Label ..... 6 - 2
Kinds of Statements ..... 6 - 3
expression Statement ..... 6 - 3
null Statement ..... 6 - 4
compound Statement ..... 6 - 4
return Statement ..... 6 - 5
if Statement ..... 6 - 5
if/else Statement ..... 6 - 6
else/if Chain ..... 6 - 7
while Statement ..... 6 - 7
do/while Statement ..... 6 - 8
for Statement ..... 6 - 8
switch Statement ..... 6 - 9
break Statement ..... 6 - 10
continue Statement ..... 6 - 10
goto Statement ..... 6 - 11

Chapter 7: The Preprocessor

File Inclusion Preprocessor Directives ..... 7 - 1
Rules for #include File Processing ..... 7 - 2
The LIBH Option and PDS Member Names Under MVS ..... 7 - 3
File Name Macro Expansion ..... 7 - 4
Conditional Preprocessor Directives ..... 7 - 4
Testing for Macro Definition ..... 7 - 4
Testing for Arithmetic Value ..... 7 - 5
Alternate Groups ..... 7 - 6
Macro Preprocessor Directives ..... 7 - 7
Macros with Arguments ..... 7 - 7
Constructing String Constants ..... 7 - 8
Replacing Arguments Inside String Constants ..... 7 - 9
Predefined Macros ..... 7 - 9
Constructing Tokens ..... 7 - 10
Removing Macro Definitions ..... 7 - 10
Information Preprocessor Directives ..... 7 - 10
Comments ..... 7 - 11
Line Control ..... 7 - 11
Pragmas ..... 7 - 11

Chapter 8: C Runtime Environment

C Library Functions ..... 8 - 1
Program Startup and Termination ..... 8 - 3
How Library Functions Indicate Errors ..... 8 - 5
How Your Program Indicates Errors ..... 8 - 7

Chapter 9: Input/Output

Streams ..... 9 - 1
Text Streams ..... 9 - 2
Binary Streams ..... 9 - 3
Buffered Input/Output ..... 9 - 4

Opening Files ..... 9 - 5
Closing Files ..... 9 - 6
Reading Streams ..... 9 - 7
Writing Streams ..... 9 - 7
Positioning Streams ..... 9 - 8
Other Stream Services ..... 9 - 9
Formatted Input/Output ..... 9 - 9
Formatted Input ..... 9 - 10
Formatted Output ..... 9 - 11

Chapter 10: Organizing Your Program
File Layout ..... 10 - 1
Function Layout ..... 10 - 3
Restrictions ..... 10 - 5
Programs with Multiple Files ..... 10 - 6
Portability Issues ..... 10 - 6

Chapter 11: C Library Reference
Header Files ..... 11 - 1
assert.h ..... 11 - 3
ctype.h ..... 11 - 5
ims.h ..... 11 - 19
limits.h ..... 11 - 23
math.h ..... 11 - 25
setjmp.h ..... 11 - 61
signal.h ..... 11 - 64
stdarg.h ..... 11 - 68
stdefs.h ..... 11 - 73
stdio.h ..... 11 - 74
stdlib.h ..... 11 - 126
string.h ..... 11 - 143
time.h ..... 11 - 164

Appendix A: Compile Time Error Messages
Errors in Your Program ..... A - 1
Environmental Problems ..... A - 21

Appendix B: Runtime Error Messages
Internal Conditions ..... B - 1
Environmental Problems ..... B - 2
Errors in the C Runtime ..... B - 3

Appendix C: Summary of Reserved Identifiers
Usage of Reserved Identifiers ..... C - 1
The Reserved Identifiers ..... C - 2

Appendix D: Glossary ..... D - 1

Index ..... X - 1

Preface

About This Manual

C Language Manual (SC09-1128) describes the C programming language and associated C library for the System/370 architecture running VM/CMS, MVS, or MVS/XA. It serves as a reference guide for programmers writing programs in the C language. This document assumes a working knowledge of programming fundamentals. Your *C Compiler User's Guide* describes how to use the C compiler on your system.

Organization of This Manual

This manual has eleven chapters.

Chapter 1, "Introduction," describes the basic features of the C language, the specific features and language extensions of C for System/370, and fundamental concepts of C program structure.

Chapter 2, "Elements of the C Language," describes the characters and symbols that you can use in a C program. It also discusses how you write the basic elements of C, and explains how to combine these language elements to form a C program.

Chapter 3, "Identifiers," describes all the things you can give names to, the scope of their names, and how their names interact.

Chapter 4, "Declarations," describes how to declare and define functions, data objects, and type definitions, and how you specify their attributes.

Chapter 5, "Expressions," describes how you call functions, compute values, and store values in data objects.

Chapter 6, "Statements," discusses the executable statements available in the C language.

Chapter 7, "The Preprocessor," covers the functions the preprocessor performs, such as file inclusion, conditional compilation, macro definition, and macro expansion.

Chapter 8, "C Runtime Environment," covers executable program structure, the nature of C library functions, and how library

functions report errors.

Chapter 9, "Input/Output," explains how you handle text and binary data streams, perform buffered input/output, and format your input and output.

Chapter 10, "Organizing Your Program," discusses common conventions, programing style considerations, and issues that affect the portability of your programs.

Chapter 11, "C Library Reference," describes each of the functions, type definitions, and macros available with the ANSI C library. For each group of functions, it describes the header file that supports that group, and then describes each of the functions that the header file declares, in alphabetical order by function name.

Appendix A, "Compile Time Error Messages," describes the error messages that the compiler may emit when it compiles your program, and how to correct the errors.

Appendix B, "Runtime Error Messages," describes the error messages that the runtime environment may emit when you run your program, and how to correct the errors.

Appendix C, "Summary of Reserved Identifiers," is a list of all identifiers reserved by the C language and declared in the header files in the C library.

Appendix D, "Glossary," defines C terms and other special terminology this manual uses.

This manual also provides an **Index.**

Related Publications

The following publications are referred to in the text of this manual. You may want to use them for more information on certain topics.

* *C Compiler User's Guide for VM/CMS*, SC09-1130.

* *C Compiler User's Guide for MVS, MVS/XA*, SC09-1129.

Chapter 1: Introduction

The C programming language is designed for general purpose use. Its key features are small size relative to other programming languages, fast execution speed, flexibility in building applications, and ease in transporting C code from one computing environment to another.

C is compact because it implements operations such as input/output and storage allocation in the form of function libraries.

C is fast and efficient because it converts many of its operators directly to machine instructions. C also features "pointers," which allow direct manipulation of the machine addresses of data objects. C further encourages efficiency and ease of maintenance with a full set of high level control flow statements and data structures. C programs often run at speeds comparable to assembly language programs, although C is far easier to code, debug, and maintain.

C is portable because it handles the majority of environmental dependencies within library functions.

Features of C for System/370

The following features are available with C for System/370:

Compilation and Execution under VM/CMS

You can compile and run your programs from CMS.

Multiple Compilation and Execution Environments under MVS and MVS/XA

You can compile your programs from a Time Sharing Option (TSO) session or from a batch Job Control Language (JCL) stream. You can run your C programs under TSO, under a standard batch initiator, or in an Information Management System (IMS) environment.

31 Bit Addressing under MVS/XA

The compiler takes advantage of the 31 bit addressing capability of MVS/XA. It supports data objects larger than 16 megabytes.

Generation of Reentrant Code

The compiler produces reentrant code for C programs.

Interfaces to Other Languages

You can write assembly language routines so that C functions may call them. You can also generate function calls that use the standard OS calling sequence employed by a number of other products.

Dynamic Linking for IMS

You can dynamically link to C programs running under IMS from other IMS programs.

Debugging Support

The compiler includes a source level debugger.

Support for Program Listings

Upon request, the compiler generates listings of C source files or combined listings of C source interspersed with assembly language code. Line numbers in the assembly language listing refer to the lines in the C source that caused the compiler to generate the assembly language code.

Automatic Register Allocation

The compiler automatically places data objects that your program references frequently in unallocated registers.

The C Language Standard

This implementation of C for System/370 is compatible with the proposed ANSI standard for the C programming language, as described in the "Information Bulletin" published by CBEMA (the April 1985 draft of the X3J11 Committee Standard).

In accordance with the proposed ANSI standard, this manual describes any behavior that is "implementation defined." When you see the phrase "on System/370," the statement that follows tells you the properties of this particular compiler. Other implementations may choose different behavior in these areas, within the limits allowed by the C language.

IBM Extensions to the Proposed ANSI Standard

The compiler supports the following extensions to the proposed ANSI standard:

Extension to the ANSI Math Library

The ANSI C math library includes the following functions (the definitions for which conform closely to the UNIX™ System V Interface Definition):

* Bessel Group functions **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**
* Erf Group functions **erf** and **erfc**
* The functions **gamma** and **hypot**.

Support For Additional Signals

The C library function **signal** supports the signals **SIGUSR1** and **SIGUSR2** as defined in the UNIX™ System V Interface Definition, in addition to those signals defined by the proposed ANSI Standard.

IMS Support

The C library includes the function **ctdli**, which permits access to an IMS data base from a C program using the data manipulation language DL/I.

Substitution of Macro Arguments Within String Constants

A compiler option is available that permits substitution of macro arguments within string constants, as some UNIX™ compilers do.

Type Conversion Options

A compiler option is available that allows your program to operate under the "unsignedness preserving" rules as UNIX™ compilers do, instead of the default ANSI "value preserving" rules.

UNIX is a registered trademark of AT&T Bell Laboratories

Storing in String Constants

In this implementation, string constants do not overlap and your program can store new values in them. This is a common extension which the proposed ANSI standard describes. The proposed ANSI standard allows overlap between string constants and prohibits storing new values in them.

Special OS Calling Sequence

The preprocessor directive **#pragma** can name functions that your program calls using a standard OS compatible calling sequence. This allows C functions to call OS compatible functions.

C Program Structure

This section describes the basic parts of every C program and how you combine these parts to build an executable file.

Functions and Data Objects

A C program source file consists of a series of printable characters. The compiler interprets these as specifications to build a set of functions and data objects. Functions become groups of machine instructions that your program can execute. Data objects become regions of memory that your program uses to store values during execution. The memory image that constitutes your executable file consists only of functions and data objects provided by your program or by the C library.

You describe the functions and data objects in your C program by "declaring" them. When you declare a function or data object, you tell the compiler about its properties. A data object declaration, for example, tells the compiler what its name is in your program, what values can be stored in it, and where in the program you may use it.

A "definition" is a special form of declaration. You must define (and also declare) any function or data object you refer to in your program. The definition of a function tells the compiler what actions to carry out when the function is called. The definition of a data object tells the compiler what initial values to store in it.

C Source Files

You present declarations and definitions of functions and data objects to the compiler in the form of a C source file. A C source file is a human readable "text" file that follows the rules for writing C. The compiler converts your source files to executable form. A program often consists of more than one C source file.

Preprocessor Capabilities

A preprocessor rewrites your C source text in several useful ways before the remainder of the compiler interprets it as described above. The preprocessor performs "directives" that you signal by writing C text lines that begin with the character #. A preprocessor directive, for example, includes the contents of another file. Others let you conditionally skip portions of your C source text. You can also define "macros" that let you write a single identifier as a symbolic name for a constant, or for a complex sequence of source text. This capability enables you to change the value of a numeric constant, for example, by changing the definition in just one place.

Compilations

A "compilation" consists of the C source file you specify when you invoke the compiler, combined with any other files that you tell the preprocessor to include. You do not necessarily provide all the source code in a single compilation. The C compiler provides an extensive library of functions. Accompanying these are a number of "header files" that you can include in your compilation to make it easier to use the library. The compiler produces an object code file for each compilation.

A C program is built by linking together the object code files necessary to define all the functions and data objects that you use. You do not necessarily provide all the object code files required to link your program. The linkage editor or loader will select object code files from the C library if they provide definitions that your object code files require. Once built, your program becomes an executable file that you can invoke like any other program on your system.

Chapter 2: Elements of the C Language

This chapter describes the characters you can use in a C source file. The compiler groups these characters into "tokens," which are the basic elements of the C language. This chapter also describes how the compiler performs this grouping. It defines every valid token and describes its use.

The Character Set for C

You write C source files using a subset of the EBCDIC character set. All the characters you need to express a C program are:

* Uppercase letters **ABCDEFGHIJKLMNOPQRSTUVWXYZ** and lowercase letters **abcdefghijklmnopqrstuvwxyz**. You use these to form identifiers and keywords. An uppercase letter is different from the corresponding lowercase letter.
* Decimal digits **0123456789**. You use these as part of identifiers, and to form integer and floating constants.
* The underscore _. You use it to form identifiers.
* The backslash \. You use it as an "escape" character, to give special meaning to the character or characters that follow.
* Punctuation **[]{}()<>!#%^&*+-=:;"',/.|~?**. You use these to form delimiters, separators, and operators. The EBCDIC "not" sign ¬ is accepted in place of the caret ^.
* Space, horizontal tab, carriage return, vertical tab, form feed, and newline. You use these "whitespace" characters to separate tokens that the compiler might otherwise interpret as one token.

Some input devices may not generate all of the characters you need to write a C program. You can replace these characters in your C source by substituting the following "trigraphs" in their places:

trigraph	*character equivalent*
??=	#
??(	[
??)	]
??/	\
??<	{
??>	}
??!	|
??'	^
??-	~

For example, you can write the C source line:

```
#define or_two(x, i)    (x[i] | x[(i) + 1])
```

as

```
??=define or_two(x, i)  (x??(i??) ??! x??((i) + 1??)).
```

Source Lines

The compiler immediately converts the sequence of source lines in your source file to a continuous stream of characters. It records the end of each source line by inserting a *newline* character in the stream. Regardless of the external format you choose for your source file, this manual describes all input in terms of this converted stream.

The compiler imposes few restrictions on how you organize your C source. In some cases, however, the newline character serves as a special delimiter. If you want to write several physical source lines where the compiler requires one source line, write a backslash character at the end of all but the last of the physical source lines. The compiler *always* discards the backslash newline sequence. It will accept a source line of up to 511 characters, counting all whitespace and the newline on the end.

Two uses of the backslash newline sequence are:

```
#define fourteen_x x + x + x + x + x + x + x + \
                   x + x + x + x + x + x + x

printf("Merely corroborative detail intended to give artistic \
verisimilitude to an otherwise bald and unconvincing narrative\n");
```

Whitespace and Comments

You use whitespace characters to separate tokens. Whitespace also helps make your program easier to read and maintain. Chapter 10, "Organizing Your Program," shows you how to use whitespace to format your source text.

Anywhere that the compiler permits whitespace, it also permits a "comment." A comment begins with the characters **/*** and ends with the characters ***/**. Between these two delimiters, you may write *any* EBCDIC characters you can edit into your source file. The comment may continue for any number of source lines. Or, you may confine it to part of a source line. A comment must end, however, before the end of the source file in which it begins. You cannot, for example, begin a comment in a source file included by the preprocessor directive **#include** and end it back in the original source file. You cannot "nest" comments, because the sequence **/*** has no special meaning within a comment. In your program a comment is equivalent to a single space character.

Some examples of comments are:

```
if (a < b)          /* a comment at the end of a line */
    a = /*(char *)*/c; /* (char *) is "commented out" */

#define abc         /* the preprocessor directive doesn't
                end until the comment does */ 17
```

Identifiers

You use identifiers to name many different things in a C program. You form all identifiers using the same rules. An identifier is a token that begins with an uppercase letter, a lowercase letter, or an underscore. Following that first character, you can write more of these characters, interspersed with decimal digits. The identifier does not end until you write some other character, such as a whitespace character or punctuation. An identifier can fill an entire source line.

When comparing two identifiers, the compiler looks at only the first 31 characters. Do not take advantage of this limitation, however. If you intend two identifiers to be the same, but spell them differently after the first 31 characters, some other compiler may treat them as distinct. Your programs will be more portable, and more readable, if you assume that all identifier characters *might* be significant.

The C library defines identifiers, for a variety of purposes, that have a leading underscore. If you choose names that have a leading underscore, you might experience conflicts with some of these names that are internal to the C library. Or, your program might work properly now, but could malfunction if you compile it with another compiler. Never use identifiers that begin with an underscore. All other identifiers are either documented in this manual as restricted or are free for you to use. If an identifier is not a keyword listed below, is not described as part of the C library, and begins with a letter, you may use it. Appendix C, "Summary of Reserved Identifiers," provides a complete list of reserved identifiers.

Some examples of valid identifiers are:

```
x
SIZE
System_370
a_rather_long_identifier
```

Keywords

A keyword is an identifier that the compiler predefines for some special purpose. The complete set of keywords in C is:

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

Constants

A constant is a token, or character sequence, that you use to represent a fixed value. There are four kinds of constants, each of which is described in detail below. These are:

* Integer constants, such as `372` or `0xFFFF`
* Floating constants, such as `10.5` or `23e-10f`
* Character constants, such as `'a'` or `'\n'`
* String constants, such as `"Look out!"`

Every constant has a "type" as well as a value. Chapter 4, "Declarations," describes types and their uses. This chapter simply names the types and states the rules for determining the type to associate with each constant. You can always tell both the type and the value of a constant just by how it appears.

Integer Constants

Integer constants represent integral numeric values. You write them in one of three bases: octal, decimal, or hexadecimal.

If the first character is a digit other than `0`, the token is a decimal integer constant, as in `129`. That digit and all digits that follow constitute a decimal number whose value is the value of the integer constant. You may follow the digits immediately with a suffix, as described below.

If the first two characters are `0x` or `0X`, the token is a hexadecimal integer constant, as in `0x3b2`. The sixteen hexadecimal digits, in order, are either `0123456789abcdef` or

0123456789ABCDEF. All such digits that follow the first two characters constitute a hexadecimal number, whose value is the value of the integer constant. You may follow the digits immediately with a suffix, as described below.

Otherwise, if the first character is **0**, the token is an octal integer constant, as in **0377**. The eight octal digits, in order, are **01234567**. All the digits that follow constitute an octal number, whose value is the value of the integer constant. You may follow the digits immediately with a suffix, as described below.

The type of an integer constant is *int* if the compiler can represent the value of the constant as an *int*. On System/370, this is a two's complement integer occupying 32 bits. Otherwise, the type is *unsigned int* if the compiler can represent the value of the constant with that type. On System/370, this is an unsigned binary integer occupying 32 bits. The constant **0xFFFFFFFF**, for example, is an *unsigned int*. If the type *unsigned int* is still not adequate, then the compiler tries to represent the number as a *long* and finally *unsigned long*, in that order. On System/370, *long* and *unsigned long* have the same representation as *int* and *unsigned int*, so they represent no additional values.

You can add a suffix to any of these forms if you want to control the type more closely. If you add **l** or **L**, the compiler chooses one of the *long* types, whether or not it needs it to represent the value. For example, the constant **10L** has type *long*. If you add **u** or **U**, the compiler chooses one of the *unsigned* types, whether or not it needs it to represent the value. The constant **0555u** has type *unsigned int*, for example. You may add the suffixes in either order. Never use the suffix **l**, since it is so easily mistaken for the digit **1**. Use **L** instead.

If you write an integer constant that the compiler cannot represent as the specified type, the compiler will emit an error message.

Integer constants are always positive. If you want to write a negative constant, use a unary minus operator, as in **-53**. This is actually a "constant integer expression," but the compiler accepts it anywhere an integer constant is permitted, and it computes the value for you at compile time. (On System/370, the expression **-2147483648** must be type cast to *int* to represent the negative integer with largest magnitude.)

Some examples of integer constants, all of which have the value 300, are

```
300
0X12C
0454
300U /* type is unsigned int */
0454L     /* type is long */
```

Floating Constants

Floating constants represent finite precision approximations to real numbers, over a range of values much larger than the integers represent. A floating constant consists of a decimal integer part, a

decimal point, a decimal fraction part, and an exponent. The exponent consists of an **e** or an **E**, followed by an optional + or - sign, followed by a decimal exponent. The decimal integer, decimal fraction, and decimal exponent are all sequences of decimal digits. You may omit either the decimal integer or the decimal fraction, but not both. You may omit either the decimal point or the exponent, but not both. The value is the decimal integer plus the decimal fraction, multiplied by ten raised to the signed decimal exponent.

The type of a floating constant with no suffix is *double*. If you add an **f** or an **F** suffix, the type is *float*. If you add an **l** or an **L** suffix, the type is *long double*. Never use the suffix **l**. You may add at most one of these suffixes. On System/370, the compiler represents the type *float* as a floating point number occupying 32 bits. The compiler represents both of the types *double* and *long double* as a floating point number occupying 64 bits.

If you write a floating constant that the compiler cannot represent as the specified type, the compiler will emit an error message.

Some examples of floating constants, all of which have the value 300, are:

```
3e2
300.
300.0
.3e3
300.0f /* type is float */
3e2L  /* type is long double */
```

All floating constants are positive. You can write a "constant expression," such as `-3.7`, however, if you want a negative floating value.

Character Constants

A character constant represents the numeric value of an EBCDIC character code or an "escape sequence." You write the EBCDIC character or escape sequence inside single quotation marks, as in `'a'` or `'\377'`. The EBCDIC character may be *any* character except backslash, newline, or a single quotation mark. You can represent these and many other character values by writing an escape sequence. The following table lists the escape sequences and the characters they generate:

escape sequence	*hexadecimal value*	*character generated*
\a	2F	alert (bell)
\b	16	backspace
\f	0C	form feed
\n	25	newline
\r	0D	carriage return
\t	05	horizontal tab
\v	0B	vertical tab
\'	7D	single quotation mark
\"	7F	double quotation mark
\?	6F	question mark
****	E0	backslash
*****ooo*		octal value
\x*hhh*		hexadecimal value

The octal value escape sequence may have up to three octal digits. An important example is **\0**, which has the value 0. This is the "null character." It is used widely in C. The hexadecimal value escape sequence may have up to three hexadecimal digits. On System/370, the compiler requires at most two hexadecimal digits to represent any character code. An example is **\x3f**. If you write a three digit hexadecimal value escape sequence, the most significant digit is lost.

You may write any EBCDIC character you can edit into your source text inside a character constant. If the character is not printable, however, your program will be hard to read and maintain. Also, if the character is not one of the set listed earlier, your program may not be acceptable to another compiler. Do not encode an octal value escape sequence or hexadecimal value escape sequence directly into your program. If you can write your string constants with just the C source characters and the escape sequences, your program will be more readable and more portable.

The value of the character constant is the value of the EBCDIC character code, or the numeric value of the escape sequence you specify. Its type is *int*. On System/370, the compiler represents *char* as an unsigned binary integer occupying eight bits, so a character constant is always positive.

String Constants

A string constant is a sequence of character codes that the compiler stores in successive locations in memory. You specify each character code in the same way as for a character constant. You write the sequence of EBCDIC characters and escape sequences inside double quotation marks, as in **"hello"** or **"That's all\n"**. Escape any double quotation mark you write in the string constant, as in **"\"Right\", he said."**. The compiler appends a null character **\0** to the end of each string constant. So the null string constant **""** occupies one character of storage

and **"a\tc"** occupies four characters of storage. The type of a string constant is *array of char*. The array size is equal to the number of characters of storage the string constant occupies.

If you use octal value escape sequences or hexadecimal value escape sequences when you write a string constant, remember that each of these includes up to three digits following the backslash. If you want to write a string constant consisting of the null character followed by the digit **1**, write it as **"\0001"**.

If you write the comment delimiter **/*** inside a string constant, as in **"/*"**, the compiler will not treat this as the start of a comment. Similarly, if you write a quotation mark inside a comment, as in

```
/* time 7hr 30' 56" */
```

the compiler will not treat this as the start of a character constant or string constant. Character constants, string constants and comments are mutually exclusive. Once you start one, the compiler stops looking for examples of the others.

If you write two string constants in a row, the compiler concatenates them to form a single string constant. There is no intervening null character, so **"abc" "def"** is equivalent to **"abcdef"**. The compiler processes escape sequences before performing the concatenation, so **"abc\0" "17def"** is equivalent to **"abc\00017def"** and not **"abc\017def"**.

String constant concatenation gives you a convenient way to break long string constants across multiple text lines. You can indent each component for readability. String constant concatenation also helps you construct string constants when you write preprocessor macros. Chapter 7, "The Preprocessor," discusses this topic in the section, "Constructing String Constants."

A string constant differs from the other constants because the compiler stores its value in a data object in memory. When you write a string constant as part of an expression, it is the *address* of the string constant that your program manipulates, not the string constant as a whole. The address "designates" the string constant. This distinction between dealing with a value and a data object designator is discussed in Chapter 5, "Expressions."

Even though a string constant is a "constant," the compiler may not always prevent you from storing new values in the string constant during program execution. Some older C programs depend on being able to store into string constants this way. For maximum portability, however, you should avoid storing into string constants. When creating a reentrant program, you must treat all string constants as read only. You cannot store new values in string constants during execution of a reentrant program.

Delimiters, Separators, and Operators

All of the remaining tokens the compiler recognizes consist of one to three punctuation characters. Some of these tokens are "operators," which you use to specify what value to compute, as in

```
x+y
```

Others are "delimiters" or "separators," which you use to group and separate other tokens, as in

```
(x+y)/z
```

or

```
return 3;
```

Sometimes the same token is an operator in one context, for example, and a separator in another. You do not need to be concerned with the distinction to understand which tokens are valid.

The following is a complete list of all the tokens made from punctuation and the names for each:

token	*names*
!	exclamation mark, logical NOT
!=	not equal to
#	pound sign
%	percent sign, modulus
%=	gets remainder
&	ampersand, address of, AND
&&	logical AND
&=	gets AND
(	left parenthesis
)	right parenthesis
*	asterisk, indirect on, multiply
*=	gets multiplied
+	plus, add
++	increment
+=	gets added
,	comma
-	minus, subtract
--	decrement
-=	gets subtracted
->	point at member
.	decimal point, dot, select member
...	ellipsis
/	slash, divide
/=	gets divided
:	colon

token	*names*
;	semicolon
<	less than
<<	left shift
<<=	gets left shifted
<=	less than or equal to
=	equal sign, gets
==	equal to
>	greater than
>=	greater than or equal to
>>	right shift
>>=	gets right shifted
?	question mark
[	left bracket
]	right bracket
^	caret, exclusive OR
^=	gets exclusive OR
{	left brace
|	vertical bar, inclusive OR
|=	gets inclusive OR
||	logical OR
}	right brace
~	NOT

The EBCDIC character set has both a "solid" vertical bar and a "broken" vertical bar. You may use either interchangeably. Parentheses **( )**, square brackets **[ ]**, and braces **{ }** can occur only in balanced pairs, as in **((a+b)+c)**.

When the compiler groups the input character stream into tokens, it always makes the longest token it can. It does so even if that grouping leads to an invalid program, and there is a different grouping of characters that is valid. For example, if you write:

```
x +++++ y
```

the compiler groups this as if you wrote

```
x ++ ++ + y
```

This is not a valid expression, while

```
x ++ + ++ y
```

might be valid in some contexts. Adopt a style of using whitespace that prevents ambiguity such as this. Chapter 10, "Organizing Your Program," describes one such style.

If the compiler sees a character it does not recognize outside of a comment, character constant, or string constant, it emits an error message. It also emits an error message if you write an invalid constant, such as **19J2**. All of your source text must contribute either to tokens, comments, or to whitespace. The rest of this manual describes the C language in terms of tokens composed by the rules described in this chapter.

Chapter 3: Identifiers

You use identifiers to name things in C. When you declare a function or data object, for example, you name it by writing an identifier as part of the declaration, as in:

```
void fun();          /* fun names a function */
int x = 3;           /* x names a data object */
```

This chapter covers many aspects of identifiers and their use. The topics it covers are:

* All the different things you must name in your programs, with a brief description of what you use them for.
* The placement of declarations, and how you declare some things implicitly.
* The concept of "name spaces." Sometimes you can use the same identifier for two different things at the same time. The rules governing name spaces tell you when you can and when you cannot do this.
* The "scope" of names. You need to know where in your source text you can use an identifier you have declared. You also need to know when you can reuse the identifier.
* The concept of "visibility" of identifiers. An identifier can be "in scope," but still not visible over portions of your program text.
* The rules for identifier linkage. Some names must be shared across compilations, while others are private to a compilation.

Naming Things in C

The following is a list of all the things that identifiers in a C program can name:

Macros: You write a **#define** preprocessor directive to define an identifier as a "macro." From that point on in your source file, the compiler replaces occurrences of that identifier with the "macro expansion" text you specify as part of the **#define** preprocessor directive. An example is:

```
#define NCHARS 511  /* NCHARS is a macro */
```

Macros are described in Chapter 7, "The Preprocessor."

Keywords: All the keywords are predefined by the compiler. You cannot add new keywords and you cannot remove any existing keywords. The names of preprocessor directives are not keywords. Keywords are listed in Chapter 2, "Elements of the C Language."

Labels: You write a label at the start of an executable statement, inside a function definition, if you want to transfer control there with a *goto* statement. For example:

```
top: x += 5;          /* top is a label */
     .....
     goto top;
```

Labels are described in Chapter 6, "Statements."

Functions and data objects: You write declarations to describe functions and data objects, and to give them names. Functions and data objects declarations are described in Chapter 4, "Declarations."

Type definitions: A "type definition" is an identifier that you can use anywhere you need to specify a type. You write declarations to describe type definitions, as in:

```
typedef char *Cptr;/* Cptr is a type definition */
```

Type definitions are described in Chapter 4, "Declarations."

Enumeration constants: An "enumeration" is a type whose values all have names. You introduce these names as "enumeration constants" when you declare the enumeration, as in:

```
enum Roman {
     I = 1, II, III,     /* enumeration constants */
     V = 5, X = 10 };
```

Enumeration constants are described in Chapter 4, "Declarations."

Tags: You can write a "tag" when you declare an enumeration type (*enum*) or a structure type (*struct* or *union*). Elsewhere in your source file, you can use the tag as a shorthand way to refer to the type. In the example above, **Roman** is an *enum* tag. Tags are described in Chapter 4, "Declarations."

Structure members: You declare "structure members" when you declare the content of a structure type. For example:

```
struct Complex {    /* struct tag */
     float re, im; };    /* structure members */
```

Structure members are described in Chapter 4, "Declarations."

Declaring Identifiers

There are several distinct contexts within a compilation. You can write the same declaration in different contexts and it will have different meanings. You can introduce identifiers for some things in one context, but not in others. The contexts are "file level," "argument level," and "block level." Below is a sample program that illustrates these various contexts. It is a complete program that you can edit into a file, compile, link, load, and run. It prints out the number of characters you type for each of the command options you specify when you invoke the program. Note that this example violates some rules of good coding practice to illustrate language features in a small example program.

```
#include <stdio.h> /* get declaration for printf */

#define SUCCESS 0  /* symbolic name for good exit status */
#define FAILURE 1  /* symbolic name for bad exit status */

enum Exit_status {      /* declare an enumeration */
     success = SUCCESS,
     failure = FAILURE};

typedef unsigned int Counter;
      /* COUNTER is a type definition */

static Counter count = 0;     /* count is a data object */

extern int main(ac, av) /* main is a function */
     int ac;            /* ac and av are arguments */
     char *av[];
     {                  /* start of function body */
     extern Counter count;   /* same count as before */
     register int i;         /* new i */

     for (i = 1; i < ac; ++i)/* look at each argument */
          {             /* start new block */
          register Counter ac = 0;/* new ac */
          register int j = 0;/* new j */

     top:               /* count characters in av[i] */
          if (av[i][j] == '\0')
               goto bottom;
          ++ac;
          ++j;
          goto top;
     bottom:
          printf("%s has %i characters\n", av[i], ac);
          count += ac;       /* accumulate total count */
          }             /* end inner block */
     printf("TOTAL: %i characters\n", count);
     exit(success);     /* terminate execution */
     }             /* end of function body */
```

This file has four "file level" declarations, one for the enumeration **Exit_status**, one for the type definition **Counter**, one for the data object **count**, and one for the function **main**. The last two declarations are also definitions. Within the definition for **main** are several other declarations. There are two "argument level" declarations, for the function arguments **ac** and **av**. A "function body" begins with the first left brace in its definition and ends with its balancing right brace. Within a function body, you can write "block level" declarations after any left brace. In this example, block level declarations occur in two places. After the first left brace that begins the function body the program redeclares **count** and declares **i** for the first time. After the second left brace, the program declares a new data object with the name **ac** and also declares **j** for the first time. In summary, the declaration contexts in this example are:

```
....
enum Exit_status {          /* file level */
     success = SUCCESS,
     failure = FAILURE};

typedef unsigned int Counter;     /* file level */

static Counter count;          /* file level */

extern int main(ac, av)        /* file level */
     int ac;                   /* argument level */
     char *av[];               /* argument level */
     {
     extern Counter count;     /* block level */
     register int i;           /* block level */
     .....
          {
          register Counter ac = 0;  /* block level */
          register int j = 0;  /* block level */
          .....
          }
     .....
     }
```

You also use declarations to write "type names." A type name is a way of writing a type without associating an identifier with the type, as in **(char)**. You use a type name to write a "type cast" operator, or to write the operand of the operator **sizeof**. These operators are described in Chapter 5, "Expressions." You can also use type names when you declare the arguments of a "function prototype." Function prototypes are described in Chapter 4, "Declarations." You can introduce identifiers in the process of writing a type name, but it is not often useful to do so. This example contains no type names.

The effect of different declaration contexts on how you name things is as follows:

Macros: To the preprocessor, a source file is just a sequence of text lines. The **#include** preprocessor directive in the

example above causes the preprocessor to include the text from the header file **<stdio.h>** as part of the compilation, but the included text is then just part of the sequence of text lines. You can write a **#define** preprocessor directive anywhere in the file, to introduce a new macro name. In the example program, the identifiers **SUCCESS** and **FAILURE** are macro names. Any additional structure is irrelevant to macro names.

Keywords: Keywords are known throughout a file. You cannot introduce any new ones. The example uses the keywords **char**, **extern**, **for**, **goto**, **if**, **int**, **register**, **static**, **typedef**, and **unsigned**.

Labels: You may only introduce labels inside a function body. In the example, there are two labels, named **top** and **bottom**.

Functions and data objects: You can declare a function at file level, at argument level, or at block level. If you declare a function at argument level, the compiler changes its type to a pointer type. You can define a function only at file level. You cannot have functions nested inside of other functions. In the example, **exit**, **main**, and **printf** are functions. The function **main** is defined by the compilation.

You can declare a data object at file level, argument level, or block level. If you declare an "array" type data object at argument level, the compiler changes its type to a pointer type. You can define a data object at file level or block level. You cannot define a data object at argument level, since that would cause the compiler to replace the argument value passed on a function call with the defining value. In the example, **ac**, **av**, **count**, **i**, and **j** are data objects. Two different data objects are named **ac**. The argument **av** is an array type that the compiler changes to a pointer type.

Type definitions: You can declare a type definition at file level or block level. You cannot declare a type definition at argument level. In the example, **Counter** is a type definition.

Enumeration constants: You can declare an enumeration anywhere a declaration is permitted, including a type name. The example contains the enumeration constants **success** and **failure**.

Tags: You can declare a tag anywhere a declaration is permitted, including a type name. In this example, **Exit_status** is an *enum* tag.

Structure members: You can declare a structure member anywhere a declaration is permitted, including a type name. There are no structure members in this example.

Name Spaces

When you declare a function, you cannot also declare a data object with the same name. Function and data object names occupy the same "name space." The interaction of various names is as follows:

Macros: Macro identifiers occupy their own name space. If you define an identifier as a macro, the compiler treats it as a macro name in all contexts, regardless of any other meaning you give to that name. You can even use keywords as macro names, although this will make your program difficult to read.

Keywords: Keywords occupy every other name space except that of macros.

Labels: Labels occupy a name space separate from all other identifiers except keywords.

Functions and data objects: The names of functions and data objects occupy the same name space as type definitions and enumeration constants. They cannot match any keywords.

Type definitions: Type definitions occupy the same name space as functions, data objects, and enumeration constants. They cannot match any keyword.

Enumeration Constants: Enumeration constants occupy the same name space as functions, data objects, and type definitions. They cannot match any keywords.

Tags: There are *enum* tags, *struct* tags, and *union* tags. All occupy the same name space. They cannot match any keywords.

Structure members: Each structure declaration introduces a separate name space. Members of a given structure must have names that differ from each other, but their names do not conflict with member names for other structures. They cannot match any keywords.

Scope of Names

Every name you introduce has a range of source text over which the compiler recognizes that name. If you write an identifier in your source file before that name is "in scope," the compiler does not recognize the connection. The same thing happens if you refer to an identifier after the name has gone "out of scope."

Many names have "block scope." A name you introduce with a file level declaration remains in scope through the end of the compilation. A name you introduce with an argument level declaration remains in scope through the end of the function body. A name you introduce with a block level declaration remains in scope through the end of the block containing the declaration.

The scope rules for names are as follows:

Macros: A macro name goes in scope when you write the **#define** preprocessor directive that defines it. It goes out of scope only if you write a **#undef** preprocessor directive that removes its definition. Otherwise, the name remains in scope through the end of the compilation. Chapter 7, "The Preprocessor", discusses preprocessor directives.

Keywords: Keywords are always in scope.

Labels: A label goes in scope when you first use it in a *goto* statement, as in **goto top;** or when you define it, as in **top: ;**. It goes out of scope at the end of the function body in which it appears. The labels in one function body do not conflict with those in another function body.

Functions and data objects: The name of a function or data object goes in scope when you declare it. You can also implicitly declare a function by naming it in a function call expression. This is described in Chapter 5, "Expressions." The names of functions and data objects have block scope.

Type definitions: A type definition goes in scope when you declare it. The names of type definitions have block scope.

Enumeration constants: An enumeration constant goes in scope when you define it as part of the enumeration that contains it. Enumeration constants have block scope.

Tags: A tag goes in scope when you first refer to it. You can use a tag to refer to an enumeration or structure even before you define its contents. Tags have block scope.

Structure members: A structure member goes in scope when you declare it as part of the structure that contains it. Structure members have block scope.

Visibility

A block structured language lets you control what names are "visible" over a region of source text. If you declare an identifier at the beginning of a block, you can mask a different declaration using the same identifier in an enclosing block. The outer identifier remains in scope, but it is no longer visible. When the newer declaration goes out of scope, the older declaration becomes visible once again. In the example program, the identifier **ac** refers to a data object of type **Counter** in the innermost block. This declaration of **ac** masks the declaration of an argument of the same name in the outer block. Any identifiers not explicitly masked in this fashion are still visible in the inner block. In the example, **av** and **count** are declared in containing blocks and used in the inner block. In fact, **count** is declared at file level, which is always the outermost block.

The effect of visibility on different names is as follows:

Macros: If an identifier is currently a macro name, you cannot give it a different definition. Macro names are always visible.

Keywords: You can mask keywords with macro definitions.

Labels: Labels are independent of block scope, so you can transfer control into a block or out of a block with a *goto* statement.

Functions and data objects: The names of functions and data objects have block scope. You can mask them by a declaration in an inner block. Argument level declarations are a special case. The compiler behaves as if all arguments are declared at the beginning of the function body. You cannot mask access to an argument by declaring something of the same name in the outermost block of a function body.

Type definitions: Type definitions have block scope. You can mask them by a declaration in an inner block. However, there are many "abbreviated" declarations that you must not use when you declare a name that is a type definition in a containing block. For instance, the delaration

```
static Counter;
```

is usually an abbreviation for

```
static int Counter = 0;
```

But if **Counter** is in scope as a type definition, the compiler assumes that you are declaring something with the base type **Counter**. It will emit an error message because you apparently declare no names.

Enumeration constants: The names of enumeration constants have block scope. You can mask them by a declaration in an inner block.

Tags: Tags have block scope, but they also present a special problem. If you want to declare two data structures that refer to each other, you must do so as:

```
struct x1 {
     struct x2 *px2;     /* point to x2 (not yet defined) */
     .....};

struct x2 {
     struct x1 *px1;     /* point to x1 (defined) */
     .....};
```

The tag **x2** lets you refer to the second structure before you write its definition. Referring to a structure you have not yet defined is necessary for describing structures that refer to each other. However, if there is a definition of **x2** in a containing block, the first structure will refer to that type instead of the one yet to come. You solve this problem by writing a special form of the *struct* declaration:

```
struct x2;          /* mask any outer meaning for x2 */
struct x1 {
     struct x2 *px2;     /* point to x2 (not yet defined) */
     .....};
struct x2 {
     struct x1 *px1;     /* point to x1 (defined) */
     .....};
```

The added declaration masks any tag **x2** in a containing block.

Structure members: The names of structure members have block scope. Visibility is not an issue, however, because the compiler determines the structure member names you can use in an expression from the type of the data object you are selecting from. For example, in the following structure declaration:

```
struct Complex {
          float re, im;
          } x;
/*  start new block  */
     {
     struct Complex {
          int re, ix;
          };
     x.re = x.im;
     .  .  .
     }
```

the structure members **re** and **im** are both of type *float*, because the type of **x** determines the names of its structure members. The fact that the tag **Complex** is not visible is irrelevant, as are the names of the structure members of the new type.

Linkage

The second declaration of **count**, at the beginning of the function body, illustrates a special kind of declaration. The declaration does not introduce a new data object. The keyword **extern**, in this context, means that the declaration is for the data object defined in a containing block. The declaration must agree with the earlier declaration for **count**, since both describe the same data object.

Two declarations are "linked" if they refer to the same function, or to the same data object. Linking is almost the opposite of masking, described under "Visibility" above. You can use block scope to introduce different data objects, for example, with the same name. Or, you can use declarations that link to declare the same data object in different scopes.

The "redeclaration" of **count** in the example was not necessary, since the identifier is known in containing blocks unless you explicitly mask it. Redeclaration is permissible, however, and often used to document what data objects outside a function body are

accessed within the function. This is an example of "internal linkage." Internal linkage is confined entirely to the compilation, and has no effect on other compilations that make up the complete program.

The example program also contains three important examples of "external linkage." The function **main** that this compilation defines is the function that the C runtime environment calls at program startup. The C runtime environment must link its declaration of **main** with this definition. On the other hand, the header file **<stdio.h>** declares the function **printf** in this compilation. The function **exit** is implicitly declared where it is called. Both declarations must be linked with the definitions provided by object code files in the C library.

When you specify that a function or data object has external linkage, the compiler modifies its name to produce an "external identifier." This identifier is passed to assemblers, linkage editors, and loaders and so follows different rules than those for identifiers internal to C. On System/370, external identifiers have seven significant characters when naming data objects, or eight when naming functions. In addition, there is no distinction between uppercase and lowercase letters. Choose your identifiers carefully, so that both internal and external name comparisons work correctly. The compiler emits an error message if distinct identifiers within one compilation produce external identifiers that are identical. The linkage editor or loader is responsible for conflicts between compilations.

Detailed rules for writing linked declarations are given in Chapter 4, "Declarations." Do not write only linked declarations of different types. Within one compilation, the compiler emits an error message for such errors. Across compilations, the linkage editor or loader generates an error message for only some errors of this kind.

A declaration may also have "no linkage." You may not write another declaration in the same block as a declaration for an identifier that has no linkage. Type definitions, enumeration constants, and function arguments always have no linkage. In the example, the data objects **i**, **ac**, and **j** declared inside the function body have no linkage.

Chapter 4: Declarations

You write declarations to describe the functions, data objects, and type definitions for your program. With each of these, you provide an identifier, a type, and other attributes. In the process of writing types, you also provide identifiers for enumeration constants, tags, and structure members. Declarations serve as a principal way to give meaning to identifiers in C.

Chapter 3, "Identifiers," describes all the different things in a C program that require identifiers. It also describes the various contexts in which you write declarations, along with such issues as name spaces, scope of names, visibility, and linkage. This chapter describes the many aspects of writing declarations, including:

* The components of a declaration: storage classes, base types, declarators, and definitions
* Storage classes
* Base types
* Declarators
* Definitions of functions and data objects
* Type definitions
* Type names.

The chapter ends with a classification of all the types used in C.

You write an "expression" as part of a declaration to specify the number of elements in an array, for example, or the initial value to be stored in a data object. Chapter 5, "Expressions," describes expressions in detail. The "constant integer expressions" referred to in this chapter are those expressions that the compiler can evaluate at compile time, to determine an integer value.

Components of a Declaration

There are four major components to a declaration:

storage class: Specifies a type definition or provides linkage information for functions and data objects. For data objects, the storage class also specifies their "lifetimes."

base type: Specifies type information common to all the declarators.

declarators: Specify an identifier, possibly with additional type information relating to that identifier. You can write multiple declarators, separated by commas, in one declaration.

definitions: Specify a function body for a function, or a data initializer for a data object. A definition, if present, immediately follows a declarator.

Some examples of declarations are:

```
static int a = 0;  /* storage class is static; type is int;
                      declarator is a; definition is = 0 */
extern int sum(int a, int b) /* storage class is extern;
                      type is int; declarator is sum(int a, int b);
                      definition follows */
    {
    return a + b;
    }
double x, y, z;     /* no storage class; type is double;
                declarators are x, y, and z; no definitions */
```

You can write the storage class anywhere among the "type specifiers" that make up the base type, but common practice is to write it first in the declaration. You can abbreviate a declaration by leaving out the storage class, if you write at least one type specifier. You can leave out the type specifiers if you write the storage class, provided you are not trying to declare a new meaning for an identifier that is in scope as a type definition. You can leave out both the storage class and the type specifiers if you are writing a file level declaration or an argument level declaration. The identifier must not be in scope as a type definition. The complete set of type specifiers is discussed later in this chapter in the section "Base Type."

One or more declarators follow the storage class and base type. Each declarator may be followed by a definition. You separate the declarators and their definitions with commas. You terminate the list of declarators and definitions with a function definition or with a semicolon.

Storage Class

The storage class, if present, consists of one of the keywords **extern**, **static**, **auto**, **register**, or **typedef**. If the storage class is **typedef**, then the declaration is a type definition. The identifier becomes a synonym for that type, and can appear anywhere a base type appears. No definition part can appear with a declarator in a type definition. You can write a type definition at file level or at block level. Some examples are:

```
typedef unsigned int Counter;      /* Counter is unsigned int */
typedef char *CHAR_PTR; /* CHAR_PTR is pointer to char */
```

For functions and data objects, the compiler interprets the storage class based on several factors:

* Whether the declaration is at file level, argument level, or block level
* Whether you are declaring a function or a data object
* Whether a declaration is in scope to which this declaration must link

Initial Declarations

If no declaration of the same identifier is in scope, then the following rules apply:

function file level declaration: If the storage class is **extern** or absent, then the function has external linkage. If the storage class is **static**, then the function has internal linkage. No other storage class is permitted. Some examples are:

```
extern int exfri();/* external linkage */
double exfrd();          /* external linkage */
static long stfrl();     /* internal linkage */
auto int aufri();  /* ERROR: invalid storage class */
```

data object file level declaration: If the storage class is **extern**, then the data object has external linkage. If the storage class is absent, then the data object has external linkage and the declaration constitutes a "tentative definition." If no explicit definition for that data object appears by the end of the compilation, then the compiler will provide an implicit definition with all zero data initializers. If the storage class is **static**, then the data object has internal linkage. Since the data object must be defined within the current compilation, any static data object declaration is a tentative definition. The section "Definitions" later in this chapter discusses tentative definitions.

No other storage class is permitted. Some examples are:

```
extern int exi;     /* external linkage */
long exl; /* tentative definition, external linkage */
static double std; /* tentative definition, internal linkage */
register int rei;  /* ERROR: invalid storage class */
```

function argument level declaration: If you declare an argument as a function type, the compiler rewrites it as a "pointer to function" type. It behaves just like any other "data object argument level declaration," described next.

data object argument level declaration: Name only identifiers that appear in the argument list for the function, and each identifier must appear at most once. If you declare an argument as an array type, the compiler rewrites it as a pointer type. If you provide no declaration for an argument identifier, the compiler implicitly declares it to be of type *int*, with the storage class absent. If the storage class is absent, then the argument is the data object passed on a function call.

An argument data object has no linkage and "dynamic lifetime." Your program allocates storage for the data object every time the function is called, and deallocates it when the function returns control to its caller. Data objects you declare with storage class **extern** or **static** have "static lifetime." The compiler determines their initial stored values from their data initializers, before program startup, and they continue to occupy storage until program termination. Program startup and termination are discussed in Chapter 8, "C Runtime Environment."

If the storage class is **register**, then the compiler tries to allocate a "register" and designate that as the argument data object. If it succeeds, your program copies the argument value passed on each function call to the register data object. You cannot take the address of a register data object, as in **&x**, so the compiler is free to implement register data objects in storage with enhanced access, such as machine registers. This restriction is true whether or not the compiler actually places the data object into a machine register. On System/370, you can place up to three data objects in registers. However, there is no limit to the type or number of data objects of storage class **register** that you can declare. All such data objects must have an integer or pointer type that the compiler represents in 32 bits. These types are described later in this chapter. If the compiler cannot allocate a register data object, it leaves the argument data object designation unchanged.

The compiler counts the number of references to each data object in a given function and puts the most frequently referenced data objects in machine registers until all three available machine registers are utilized. The compiler performs all explicit requests for registers made by declaring data objects of storage class **register** *before* performing automatic register allocation.

No other storage class is permitted. Some examples are:

```
int f(a, b, c, d)
     double c; /* third argument is double */
     register int a;     /* register requested for first */
     long b;   /* second is long */
     {.....}        /* fourth is int by default */

double d(x, y, z)
     double x, y;   /* x and y are double */
     double x;      /* ERROR: x is declared twice */
     double w;      /* ERROR: w is not an argument */
     static double z;    /* ERROR: invalid storage class */
```

function block level declaration: For any storage class, the function has external linkage. Some examples are:

```
int f() {
    extern int g();     /* external linkage */
    double h();    /* external linkage */
```

data object block level declaration: If the storage class is **extern**, then the data object has external linkage. If the storage class is **static**, then the data object has no linkage, static lifetime, and is a tentative definition. The identifier is visible only inside the block. No other declaration can be linked to this one.

If the storage class is **auto** or absent, then the data object has no linkage and dynamic lifetime. The compiler allocates storage for the data object every time control enters the block during execution of the containing function, and deallocates it when control leaves the block.

If the storage class is **register**, then the compiler tries to allocate a register for the data object. Such a register data object has no linkage and dynamic lifetime. You cannot take the address of a register data object. If the compiler cannot allocate a register, it allocates the data object just as for the **auto** storage class.

Some examples are:

```
int f()
    {
    extern int exi;     /* external linkage */
    static long stlo;   /* no linkage, static lifetime */
    auto double audo;   /* no linkage, dynamic lifetime */
    float aufl;         /* no linkage, dynamic lifetime */
    register int rei;   /* no linkage, register requested */
```

Redeclarations

If a declaration of the same identifier is in scope, then the declaration is a "redeclaration," and the following rules apply:

function file level redeclaration: If the storage class is **extern** or absent, then the function inherits linkage from the previous declaration. If the storage class is **static**, then the function must have internal linkage from the previous declaration. *If you want a function to be known only within the current compilation, use* **static** *storage class in its first declaration.* Subsequent declarations can be **extern** for any function declaration.

No other storage class is permitted. Some examples are:

```
extern int exfri();
extern int exfri();/* external linkage inherited */
static int exfri();/* ERROR: wasn't internal linkage */

static double stfrd();
external double stfrd();/* internal linkage inherited */
static double stfrd();  /* internal linkage consistent */
```

data object file level redeclaration: If the storage class is **extern**, then the data object inherits linkage from the previous declaration. If the storage class is absent, then the data object inherits linkage from the previous declaration and the combined declaration becomes a tentative definition. If the storage class is **static**, then the data object must have internal linkage from the previous declaration.

No other storage class is permitted. Some examples are:

```
extern int exi;
extern int exi;     /* external linkage inherited */
int exi;       /* external linkage inherited */

static long exlo;
extern long exlo;  /* internal linkage inherited */
```

function argument level redeclaration: This is not permitted.

data object argument level redeclaration: This is not permitted.

function block level redeclaration: For any storage class, the function inherits linkage from the previous declaration. Some examples are:

```
extern int exfri();
static double stfrd();
g() {
     extern int exfri();/* external linkage inherited */
     double stfrd();          /* internal linkage inherited */
```

data object block level redeclaration: If the storage class is **extern**, and the previous declaration specifies internal or external linkage, the data object inherits that linkage. If the previous declaration specifies no linkage, you are declaring a different data object from the one whose name is currently in scope. Similarly, if the storage class is anything other than **extern**, you are also declaring a different data object. The compiler will emit an error message if you declare different data objects with the same identifier within the same block.

Some examples are:

```
extern int exi;
static long stlo;
static double stdo;/* ORIGINAL stdo */
int g() {
     extern int exi;     /* external linkage inherited */
     extern long stlo;   /* internal linkage inherited */
     auto double stdo;   /* NEW: no linkage */
          {
          extern int exi;     /* same exi as always */
          extern double stdo;/* ORIGINAL */
          auto long aulo;
          extern long aulo;   /* ERROR: different */
```

The visibility of **stdo** in this example is quite complex. It is first declared at file level with internal linkage and static lifetime. It is then masked by the **auto** declaration at the top of the function which has no linkage. The **extern** declaration in the inner block links to the declaration at file level. The original **stdo** is unmasked for the duration of the inner block, then masked once again until the end of the function.

There are many ways to control visibility and linkage in C, but the rules are complex. The keywords **extern** and **static** are "overloaded." They have very different meanings in slightly different contexts. You can avoid most of the subtle cases described here quite easily, however. Declare all functions and all data objects with external linkage, at file level. Declare all other file objects in the innermost block with sufficient scope for all references. Do not intentionally redeclare external identifiers. Treat them as "reserved identifiers" throughout your program. Use block scope to minimize accidental name conflicts among identifiers. Do not use block scope to introduce redefinitions that harm readability. You need to use only a few of the many forms of declarations detailed here. Chapter 10, "Organizing Your Program," discusses style restrictions that help you write better C code.

Base Type

The "type" you write in a declaration characterizes many aspects of the thing being declared. Types fall into three major groups:

function types: You distinguish functions by declaring a type of the form *function returning T*, where *T* is a type that a function may return. All such types are called "function types."

incomplete types: You give an expression that produces no value the special type *void*. If you do not provide enough information so the compiler knows the size of a data object, you have a type that describes a "data object of unknown content." These types plus *void* constitute the "incomplete types."

data object types: All other types other than function types and incomplete types are "data object types."

Each data object type has a set of "values" that it can represent. These values are different bit patterns of the "representation" of that data object type. The compiler knows exactly how many bits to allocate for a data object of a given type. The C language permits enough variation in the representation of each type that efficient encoding is possible on many different computer architectures. So the type of a data object tells you how much storage the compiler allocates for it, what bit patterns your program may store in the data object and what value to assign to each valid bit pattern when your program reads the contents of that storage.

The "base type" is that part of the type information you write as the first part of the declaration. Each declarator in the declaration may provide additional type information, but all share the base type. For example,

```
int in, *pi, fri(), ai[];
```

declares **in** to be of type *int*, **pi** to be of type *pointer to int*, **fri** to be of type *function returning int*, and **ai** to be of type *array of int*. The base type for this declaration is *int*, specified by the keyword **int**.

You write a base type by writing one or more "type specifiers" that describe the type. You may write the type specifiers in any order, and the storage class may appear anywhere among the type specifiers. For example, to declare the variable **x** with the storage class **static** and the type *long double*, you can write any of:

```
static long double x;
static double long x;
long double static x;
long static double x;
double static long x;
double long static x;
```

Always use the first form, however, since it is the most readable. Types written with multiple type specifiers are presented here in the forms most widely used. If you can write a type more than one way, use the form listed first.

All base types fall into one of the following categories:

* The "void type:" *void*
* The "integer types:" *char*, *signed char*, *unsigned char*, *short*, *unsigned short*, *int*, *unsigned int*, *long*, and *unsigned long*
* The "floating types:" *float*, *double*, and *long double*
* The "composite types:" *enum*, *struct*, and *union*

The type *void* is always an incomplete type. The composite types may be either incomplete types, if their content is not yet specified, or data object types.

A base type may be qualified by either or both of the "type qualifiers" **const** and **volatile**. All base types and qualifiers are

described in order, along with their representations on System/370. Several types often have the same representation. Nevertheless, each of the types listed here is considered to be different, when the compiler checks if two types are the same.

All of the base types describe data objects that occupy a integral number of bytes of contiguous storage. There are no unallocated regions of storage within a data object, and there are no bytes that are only partially filled. There may be "padding" bits, which do not contribute to the value of any component of a composite data object. These are still part of the value of the data object as a whole. A byte in C always consists of at least eight bits. On System/370, a byte consists of exactly eight bits.

All of the base types describe data objects that begin on a byte boundary. Each type may also require a stronger "alignment" in storage. The data object may be constrained, for example, to begin on an even numbered byte address. The compiler enforces any alignment required for data objects when it allocates storage for them. For a composite data object, it ensures that each component is adequately aligned and that the data object as a whole is aligned as strictly as its most demanding component. System/370 *requires* no special storage alignment, but *favors* access to a data object aligned on a data address that is a multiple of its size in bytes. The compiler aligns data objects on storage boundaries favored by System/370.

The Void Type

You write type *void* as **void**. This type has no representation. It is the only incomplete type that cannot be completed. You cannot declare a data object of type **void**, as in

```
void x;   /*   ERROR     */
```

Valid types based on *void* are *pointer to void* type, *function returning void* types, and other types derived from these.

The Integer Types

The integer types each represent a range of integers. Some are "signed integers," which represent a range including both negative and positive values. Others are "unsigned integers," which represent a range from zero to some positive value. All integers in C have a binary representation.

char type

You write type *char* as **char**. The type is an integer that occupies one byte of storage. Each compiler chooses whether to represent type *char* as a signed or an unsigned integer. Regardless of this choice, all of the character codes you use to write C programs are positive values when represented as type *char*. Chapter 2, "Elements of the C Language," describes these characters. You use data objects of type *char* to hold character codes. On System/370, the compiler represents a data object of type *char* as an unsigned binary integer, with values from 0 to

255, inclusive.

signed char type

You write type *signed char* as **signed char**. The type is a signed integer that occupies one byte of storage. You use data objects of this type to hold small signed values. On System/370, data objects of type *signed char* can hold values from −128 to 127, inclusive.

unsigned char type

You write type *unsigned char* as **unsigned char**. The type is an unsigned integer that occupies one byte of storage. You use data objects of this type to hold small unsigned values. On System/370, data objects of type *signed char* can hold values from 0 to 255, inclusive.

short type

You write type *short* as any of: **short**, **short int**, **signed short int**, or **signed short**. The type is a signed integer that occupies at least 16 bits of storage. You use data objects of this type to hold moderate signed values. On System/370, the type *short* occupies 16 bits of storage. Its values range from −32,768 to 32,767, inclusive.

unsigned short type

You write type *unsigned short* as either of: **unsigned short** or **unsigned short int**. The type is a unsigned integer that occupies at least 16 bits of storage. You use data objects of this type to hold moderate unsigned values. On System/370, the type *unsigned short* occupies 16 bits of storage and can hold values from 0 to 65,535, inclusive.

int type

You write type *int* as **int**. If you specify no base type, the compiler assumes you mean the type *int*. You can also write the type *signed int* as **signed int**. The compiler treats the type *signed int* as identical to plain *int*, except when you declare a bitfield. Bitfields are discussed later in this chapter. The type *int* is a signed integer that occupies at least 16 bits of storage. You use data objects of this type to hold moderate signed values. The C language *strongly* favors arithmetic with type *int* or *unsigned int*. On System/370, the type *int* actually occupies 32 bits of storage. Data objects of type *int* can hold values from −2,147,483,648 to 2,147,483,647, inclusive.

unsigned int type

You write type *unsigned int* as either of: **unsigned int** or **unsigned**. The type is an unsigned integer that occupies at least 16 bits of storage. You use data objects of this type to hold moderate unsigned values. The C language *strongly* favors

arithmetic with type *int* or *unsigned int*. The type *unsigned int* actually occupies 32 bits of storage. On System/370, data objects of type *unsigned int* can hold values from 0 to 4,294,967,295, inclusive.

long type

You write type *long* as any of: **long**, **long int**, **signed long**, or **signed long int**. The type is a signed integer that occupies at least 32 bits of storage. You use data objects of this type to hold large signed values. On System/370, the type *long* has the same representation as *int*.

unsigned long type

You write type *unsigned long* as either of: **unsigned long** or **unsigned long int**. The type is an unsigned integer that occupies at least 32 bits of storage. You use data objects of this type to hold large unsigned values. On System/370, the type *unsigned long* has the same representation as *unsigned int*.

The Floating Types

The floating types each represent a range of finite precision approximations to the real numbers. Each range may be different, but each includes both negative and positive values plus the value zero. All must maintain at least six decimal digits of precision over a range from ten to the power −38 to ten to the power +38.

float type

You write type *float* as **float**. This is the floating type that occupies the least amount of storage. You use data objects of this type to hold floating values that may be of the minimum required precision. On System/370, the type *float* is a floating point data type that occupies 32 bits of storage. It retains at least six decimal digits of precision, over a range from approximately **1E−76** to **1E+76**.

double type

You write type *double* as **double**. This is a floating type that should retain greater precision than the type *float*. The C language *strongly* favors arithmetic with type *double*. You use data objects of this type to hold floating values that should be of the maximum precision normally available. On System/370, the type *double* is a floating point data type that occupies 64 bits of storage. It retains at least 15 decimal digits of precision, over a range from approximately **1E−76** to **1E+76**.

long double type

You write type *long double* as **long double**. This is a floating type that retains the maximum precision available during *intermediate* calculations of floating expressions. You use data objects of this type to hold floating values that should be of extraordinary precision. On System/370, the type *long double* has the same representation as the type *double*.

The Composite Types

The three composite types **struct**, **union**, and **enum** share a common format. The type descriptor for any of these consists of the proper keyword, followed by a tag, followed by the "content" inside braces. If you write both the tag and the content, you can later use the tag alone to refer back to the content. If you write just the content, you have an "unnamed" composite type. Only identifiers declared within the declaration having this base type can have the same unnamed composite base type. If you write just the tag, you can refer either forward or back in your source text to the content that goes with it. When you use a tag to talk about a composite type whose content you have not yet written, you are specifying an incomplete type. You can use an incomplete type anywhere that the compiler does not need to know the size of data objects of that type.

Since *struct* types and *union* types share so many properties, this manual uses the term "structure" frequently to refer to both. An incomplete *struct* or *union* type, for example, is a *structure of unknown content*. You can also write an *enum of unknown content*, but that has fewer uses.

Here are some examples of the use of tags:

```
struct Complex {          /* both tag and content */
    float re, im; } x; /* x is of type struct Complex */
union mix *p;  /* union mix has unknown content,
                  p is of type pointer to union mix */
struct Complex y;         /* y is of same type as x */

enum {RED, YELLOW, GREEN} light;  /* unnamed type */
```

There is a special way to declare an incomplete composite type when a tag of the same name is in scope in a containing block. You write a declaration with no declarators, as in:

```
union mix;      /* introduce new tag "mix" in this block */
union mix *p;  /* p points to the new "mix" */
```

Chapter 3, "Identifiers," discusses this issue, in the section "Visibility."

Enumeration Types

You write an "enumeration" type as the keyword **enum**, followed by a tag, followed by the content inside braces. You may omit either the tag or the content, but not both. An enumeration is an integer type whose values you name when you write the content.

You use data objects of enumeration type to hold a small number of distinct values. The compiler chooses one of the integer types other than *long* or *unsigned long* to represent each enum. If all of the values you specify in the content of the enumeration can be represented in a *signed char*, for example, the compiler will represent the enumeration as that type. When the compiler compares two types, it uses the integer type it chose as the type of the enumeration. You may store values in a data object of enumeration type other than those values you define for the enumeration. If you do, make sure that the values you store are representable in the chosen type.

The content of an enumeration consists of one or more enumeration constant declarations, which you separate with commas. An enumeration constant declaration consists of an identifier, or an identifier followed by an equal sign, followed by a constant integer expression giving the value of the enumeration constant. If you write only an identifier, the value of the enumeration constant is zero for the first enumeration constant declaration in the list. For later declarations in the list, the value is one more than the value for the preceding declaration. An enumeration constant declaration defines its identifier to be an enumeration constant. Its type is the type the compiler chose to represent the enumeration and its value is the enumeration constant value for that declaration. You can use enumeration constants almost anywhere you can use any of the integer constants described in Chapter 2, "Elements of the C Language". If you use them in **#if** of **#elif** preprocessor directives, however, the compiler will treat them as undefined macro names, not enumeration constants.

Some examples of enumerations are:

```
enum Roman {
     I = 1, II, III,          /* values are 1, 2, 3 */
     V = 5, X = 10 };    /* values are 5 and 10 */
enum {
     OPEN, CLOSE, READ, WRITE,     /* values are 0, 1, 2, 3 */
     OP = 0, CL, RE, WR};     /* values are also 0, 1, 2, 3 */
```

You may define enumerations that declare multiple enumeration constants with the same value, as the last example illustrates.

Struct Types

You write a "struct" type as the keyword **struct**, followed by a tag, followed by the content inside braces. You may omit either the tag or the content, but not both. A data object of struct type is composed of one or more "structure members" that occupy sequentially increasing addresses in storage. The first structure member is a data object at offset zero from the start of the composite data object. Each subsequent structure member begins on a storage boundary, following the previous structure member, that is suitably aligned for its type. The compiler may provide padding between structure members to enforce proper alignment. The compiler may also provide padding after the last structure member so that a data object of the same type can immediately

follow and be on a proper storage boundary, as in an array of structures. You use a struct type to declare a collection of related data objects that you wish to manipulate as a single data object.

The content of a struct type consists of a sequence of declarations. Each declaration names one or more structure members. Do not specify a storage class as part of any declaration. Declare only data object types. Also, do not specify a definition for any declarator. Some examples are:

```
struct address {
     char name[50]; /* first member is array of char */
     char street[50];    /* array of char */
     char town[30]; /* array of char */
     long zip_code; /* long integer */
     };
struct node {
     struct node *prev; /* root.prev->value is previous value */
     struct node *next; /* root.next->value is next value */
     long value;              /* root.value is current value */
     } root;             /* root of binary tree */
```

The last example illustrates that you can declare a struct type that contains pointers to objects of the same type, by declaring a *pointer to struct of unknown content*. The compiler does not have to know the size of the whole struct, since it is just adding a pointer to the struct.

Bitfields

You may write a "bitfield" declarator as part of the content of a structure type. A bitfield is a contiguous group of bits within one structure member of type *int*. You declare a bitfield by writing a declaration that has base type *signed int*, *unsigned int*, or *int*. Each bitfield declarator consists of an identifier followed by a colon, followed by a constant integer expression giving the size in bits of the bitfield. You can declare an "unnamed bitfield" by leaving out the identifier. The size in bits must be a value between zero and the number of bits used to represent the type *int*, inclusive. On System/370, an *int* data object occupies 32 bits of storage.

You may declare one or more bitfields in sequence. The compiler adds a new structure member of type *int* for the first bitfield in a sequence. If the compiler can pack each subsequent bitfield into the same structure member, it uses the group of bits adjacent to those used for the previous declarator. On System/370, the first bitfield in a sequence occupies the most significant bits of the *int*. Subsequent bitfields pack into the most significant bits remaining in that same *int*. The compiler adds a new structure member if it cannot pack a bitfield into the current structure member, or if you specify a bitfield of width zero. A bitfield of width zero must be an unnamed bitfield. You can also use an unnamed bitfield anywhere you want to control padding between named bitfields.

The compiler determines the type of each bitfield from the base type of its declaration. If the base type is *signed int*, then the bitfield is a *signed bitfield* type. The compiler interprets it as a

two's complement integer, so it can represent a range including both negative and positive values. If the base type is *unsigned int*, then the bitfield is an *unsigned bitfield* type. The compiler interprets it as an unsigned binary integer, so it can represent a range of values beginning with zero. If the base type is *int*, then the bitfield is a plain *bitfield* type. Its representation is the same as either *signed bitfield* or *unsigned bitfield*. On System/370, the representation is the same as for an *unsigned bitfield*. This is the only context where the compiler distinguishes the type *signed int* from plain *int*. The bitfield types are all integer types.

Some examples of bitfields are:

```
struct old_ptr {
     int : 8; /* most significant 8 bits unnamed */
     int pointer : 24; };     /* remaining 24 bits */
struct answers {
     int smdw : 2, /* x.smdw */
          age : 7, /* x.age */
          can_drive :1, /* x.can_drive */
          can_type :1; } x;  /* x.can_type */
```

Union Types

You write a "union" type as the keyword **union**, followed by a tag, followed by the content inside braces. You may omit either the tag or the content, but not both. A data object of union type is composed of one or more structure members that overlap in storage. All structure members are data objects at offset zero from the start of the composite data object. The composite data object begins on a storage boundary that is suitably aligned for any of its structure members. The compiler provides padding after any structure members smaller than the entire composite data object. You use a union type to declare a data object that may hold data objects of different types at different times.

The content of a union type consists of a sequence of declarations. Each declaration names one or more structure members. The rules for writing structure member declarations for a union type are the same as those for a struct type. If you specify a sequence of bitfields, the compiler treats each as the first of a sequence. On System/370, every bitfield in a union type occupies the most significant bits of a 32 bit integer. Some examples are:

```
union arith {
     int ival;      /* may contain an int */
     double dval;   /* or a double */
     };

union {
     char *cp;      /* point to char with ptr.cp */
     short *sp;          /* or to short with ptr.sp */
     int *ip;       /* or to int with ptr.ip */
     } ptr;
```

Only one structure member of a union data object may have a valid stored value at any given time while your program is

executing. Your program must determine which structure member has a valid stored value, since the union data object contains no such indication. You can store a value in one structure member of the union and access it properly as another structure member of the union, provided both structure members have the same type. If two structure members of a union are struct types whose first structure members have the same types, then you can store a value in one of those structure members and access it properly as the corresponding structure member of the other struct in the union. For example:

```
union {
     struct {
          int utag; /* utag is common to both structs */
          double val;    /* holds double x.d.val */
          } d;
     struct {
          int utag;
          int val;  /* holds int x.i.val */
          } i; } x;
```

You may store a value in **x.i.utag** and test it properly by accessing **x.d.utag**.

Except for this situation, if you store a value in one structure member of the union and access it as a structure member of the union having a different type, your program may work quite differently when compiled by different compilers. Avoid such operations if you want your program to be portable.

Type Qualifiers

You can add the type qualifiers **const** and **volatile** to any base type. You can also add them to pointer type attributes, which are discussed later in this chapter. Each type qualifier can appear at most once in a base type or a pointer type attribute. You may specify **const** and **volatile** together, in either order.

The Type Qualifier const

When you add the type qualifier **const** to any base type, you make it a "const" type. If you write an expression that obviously alters the value stored in the data object, the compiler will emit an error message. Do not write an expression that covertly alters the value stored in the data object. You use const types to declare data objects whose stored values you do not intend to alter during execution of your program.

If you declare a data object of const type with static lifetime, either at file level or at block level, you may specify its stored value by writing a data initializer. The compiler determines its stored value from its data initializer before program startup, and the data object continues in existence until program termination. If you specify no data initializer, the stored value is zero. If you declare a data object of const type at argument level, you tell the compiler that your program will not alter the value stored in that argument data object by the function call. If you declare a data

object of const type and dynamic lifetime at block level, you may specify its stored value by writing a data initializer. If you specify no data initializer, the stored value is indeterminate. The value you specify will be stored in the data object upon every transfer of control to the beginning of the block. Chapter 6, "Statements", describes transfer of control.

If you declare a structure member to be a data object of const type, you tell the compiler that your program will not alter the value stored in that structure member. If the structure type is a const type, then your program may not alter the value stored in any of its structure members.

Some examples of const types are:

```
const float pi = 355.0 / 113.0;   /* pi never changed */
int f(arg)
     const double arg;  /* arg never changed inside f */
     {
     const double sin_arg = sin(arg);
                    /* sin_arg never changed after this */
```

The Type Qualifier volatile

When you add the type qualifier **volatile** to any base type, you make it a "volatile" type. An expression that stores a value in a data object of volatile type stores the value immediately. Your program will not defer or eliminate storing a value in such a data object. An expression that accesses a value in a data object of volatile type obtains the stored value for each access. Your program will not reuse the value accessed earlier from such a data object. You use volatile types to declare data objects whose stored values may be changed in ways that the compiler cannot anticipate.

Most applications have no need for volatile types. You use volatile types to declare data objects that appear to be in conventional storage but are actually represented in machine registers with special properties. You use them to declare data objects that are in storage shared among multiple programs. You can also use them to declare data objects that "signal handlers" may access. A signal handler is a function you specify that responds to events that might not be synchronized with the execution of your program. Signal handlers are described with the header file **<signal.h>** in Chapter 11, "C Library Reference."

Declarators

You write a declarator to specify the identifier you are declaring, and to provide additional type information. If the base type is not an enumeration or a structure type, you must specify at least one declarator. After the declarator you can write a function definition or a data initializer. If you follow a declarator with a function definition, you terminate the declaration. Otherwise, you can then write a comma followed by another declarator. If you do not terminate the declaration with a function definition, you write a

semicolon to terminate the declaration.

The simplest declarator you can write is an identifier alone, such as **x**. This declares the identifier **x** to have the base type. To make more complex declarators, you can add three kinds of "type attributes" to an identifier **x**, whose base type is *T*:

***x:** declares **x** to be of type *pointer to T*

x[size]: declares **x** to be of type *array of T*, with the number of elements specified by *size*

x(args): declares **x** to be of type *function returning T*, with its argument types specified by *args*

If you specify more than one of these type attributes, you read them from right to left. For example, ***fp()** is of type *function returning pointer to T*, for base type *T*, and ****app[5]** is of type *array of pointer to pointer to T*. You can also use parentheses to specify that type attributes read in a different order. You read the contents of the innermost parentheses first. For example, **(*pf)()** is of type *pointer to function returning T*, and ***(*pap)[5]** is of type *pointer to array of pointer to T*.

These rules model the way the compiler applies the addressing operators when you write expressions. You write the *pointer* type attribute in a declarator the same way you write the "indirect on" operator in an expression. You write the *array* type attribute the same way you write a "subscript" in an expression. You write the *function* type attribute the same way you write a "function call" in an expression. You also read the type attributes in the same order as the compiler "groups" the corresponding operators in an expression. These operators are described in Chapter 5, "Expressions."

If you wish to change a declarator of type *T* to type *pointer to T*, replace the identifier **x** within the declarator with **(*x)**. If you want to change the declarator to type *array of T*, replace the identifier with **(x[])**. If you want to change the declarator to type *function returning T*, replace the identifier with **(x())**. You may not always need the parentheses, but using them will not cause an error.

Below are the rules for applying each of the type attributes to an identifier alone. You build more complex declarators by applying the rules above.

Pointer Type Attribute

You write the pointer type attribute as an asterisk *****, followed by any type qualifiers, followed by the identifier.

If *T* is the base type, the type of this declarator is *pointer to T*. *T* may be *any* type. Do not take the type *pointer to void* literally, however. It is used as a special form of universal data object pointer, as described in Chapter 5, "Expressions." You use pointers to hold the addresses of functions or data objects. On System/370, all pointers occupy 32 bits of storage and support 31 bit addresses on MVS/XA or 24 bit addresses on other systems.

You write *** const cp** to declare that **cp** is a *const pointer to T*. You write *** volatile vp** to declare that **vp** is a *volatile pointer to T*. You can also write both type qualifiers after the *****, in either order. Type qualifiers written after the ***** describe the pointer data object, while type qualifiers written with the base type describe the data object pointed to. Note the four distinct cases:

```
char *p;                  /* pointer to char */
const char *pc;           /* pointer to const char */
char * const cp;          /* const pointer to char */
const char * const cpc; /* const pointer to const char */
```

Array Type Attribute

You write the array type attribute as the identifier followed by an array size in brackets **[ ]**. If *T* is the base type, the type of this declarator is *array of T*. *T* must be a data object type. If the array size is absent, the type is an *array of unknown content*. This is an incomplete type. Otherwise, the array size must be a constant integer expression whose value is greater than zero. The array size gives the number of elements, or instances of a data object of type *T*, within the array. You use an array to hold an ordered sequence of data objects all of the same type.

You can use an *array of unknown content* in several contexts. If you provide a data initializer with the declarator, the compiler sets the size to the number of array element data initializers you write. Two examples are:

```
int x[] = {1, 2, 3};          /* size is 3 */
char mesg[] = "hello";  /* size is 6 (including null) */
```

If you write a declarator that is not a defining instance, you can declare it as an *array of unknown content*. You can use identifiers declared this way to access elements of the array. An example is:

```
extern double tab[];     /* tab defined elsewhere */
```

If you declare a pointer to an array, the array can be an *array of unknown content*. You may not use the pointer to access storage, however. For example:

```
char (*pa)[]; /* pointer to array of char */
```

If you declare a function argument to be an array, the compiler rewrites the declaration as a pointer to the first element of the array. This is a complete type. So the array can be of unknown content. For example:

```
int main(ac, av)
     int ac;
     char *av[];    /* actually passed as char **av */
```

If you redeclare an array, either of the declarators may omit the size. If both provide a size, the two sizes must be the same. Once you provide a size, the compiler remembers it as part of the combined declarations. It becomes a data object type, and is no longer an incomplete type. For example:

```
extern char extc[];/* unknown size */
char extc[25];     /* compatible with first */
extern char extc[25];   /* compatible with first two */
extern char extc[];/* still compatible */
char extc[20];     /* ERROR: size still known */
```

Function Type Attribute

You write the function type attribute as the identifier followed by an argument list in parentheses **( )**.

If *T* is the base type, the type of this declarator is *function returning T*. *T* must be a data object type other than *array of T*, or the type *void*. You use functions to express *all* of the executable code you supply with your program.

You can write several kinds of argument list. These represent combinations of two choices you make:

function definition: If you provide a definition for the function with this declaration, you must provide the names of the function arguments in the argument list.

function prototype: If you want the compiler to check the types and number of arguments on subsequent function calls and redeclarations, you must provide the types of the function arguments in the argument list. A function declaration that provides argument types is called a "function prototype."

Some examples are:

```
int fri();              /* no definition, no prototype */
int fri(int, double);   /* no definition, prototype */
int fri(int a, double d);    /* no definition, prototype */
int fri(x, y)       /* definition, no prototype */
    int x;
    double d;
    {.....}
long frlo(long lo) /* definition, prototype */
    {.....}
```

If a declaration is not a function prototype or a definition, its argument list must be empty. This is *not* an incomplete type. An incomplete type is a data object type that is missing some information that the compiler needs to know before it can determine the size of objects declared with that type.

If a declaration is not a function prototype but is a definition, an empty argument list means that the function has no arguments. Otherwise, write the argument identifiers alone within the argument list. You declare any arguments with argument level declarations immediately following the declarator. The function body, enclosed in braces, terminates the argument level declarations. You may declare each argument at most once. You may declare only arguments at argument level. If you do not declare an argument, the compiler implicitly declares it to have type *int*.

If a function prototype is not also a definition, you may omit the argument identifiers. An argument declaration of this form is

called a "type name" and is discussed later in this chapter. If you supply argument identifiers, their scope ends with the end of the argument list. They need not match identifiers you use in redeclarations of the function.

If a function prototype is also a definition, provide the argument identifiers along with their types. The scope of these argument identifiers ends with the end of the function body.

If you write an argument type of the form *function returning T*, the compiler changes it to *pointer to function returning T*. If you write an argument type of the form *array of T*, the compiler changes it to *pointer to T*. Otherwise the type you write must be a data object type.

Function Prototype Arguments

You write a function prototype for a function **f** with no arguments as **f(void)**. For a function with a fixed number of arguments, you write a list of declarations, one for each argument. You separate the declarations with commas. Each declaration can have only one declarator. A function prototype argument declaration can have the storage class **register**, which has meaning only if the prototype is also a definition. A function prototype argument of "const" or "volatile" type has meaning only if the function prototype is also a definition. The compiler checks the types and number of arguments on a function call in scope of a prototype. Chapter 5, "Expressions" describes the rules for calling functions in the section "Function Calls."

If you want to call a function with a variable length argument list, write a function prototype whose argument list ends with a comma followed by the ellipsis, as in:

```
int printf(char *fmt, ...);
```

You must specify at least one argument in the function prototype. On a function call, the compiler checks the types and number of all the arguments that you specify in the function prototype. You may call a function with more arguments than you specify in the function prototype, but not with fewer. All function calls and the definition must be in scope of such a function prototype, if you want your program to be highly portable.

If you redeclare a function, either of the declarators may be a function prototype and the other not a function prototype. If both provide a function prototype, the number of arguments and their types must be the same. Chapter 5, "Expressions," describes when types are the same in the section "Comparing Types". Once you provide a function prototype, the compiler remembers it as part of the combined declarations.

OS Calling Sequence

If you wish to call a function that uses the standard OS calling sequence, such as a function written in another language, declare the function in a special way. Chapter 7, "The Preprocessor," describes how to specify that a function uses the OS calling

sequence, in the section "Pragmas."

Definitions

You write a definition immediately after a declarator. If the declarator declares an identifier of function type, the definition must be a sequence of argument declarations followed by a function body. If the declarator declares an identifier of data object type, or an *array of unknown content*, the definition must be a data initializer. In either case, the identifier is declared at the end of the declarator. It is in scope for the definition. A function can therefore call itself and a data object can use its own address as part of its data initializer. Do not write a definition for a declarator whose type is incomplete, except for an *array of unknown content* as described above.

Function Definition

You define a function by specifying its argument declarations and function body. You also provide a "defining instance" of the function identifier when you specify a function body. Every function your program makes use of must have exactly one defining instance. That defining instance may be in one of the compilations you provide, or it may be in the C library the compiler provides. If you declare a function with storage class **static** at file level, it has internal linkage. You must provide a defining instance in the same compilation. Calling a function is described in Chapter 5, "Expressions," in the section "Function Calls". Writing the body of a function is described in Chapter 6, "Statements."

Data Object Definition

You define the value stored in a data object by specifying a data initializer. You also provide a "defining instance" of the data object identifier when you specify a data initializer. Every data object your program makes use of must have exactly one defining instance. That defining instance may be in one of the compilations you provide, or it may be in the C library the compiler provides.

If a data object has dynamic lifetime, its one and only declaration is its defining instance. In this case, your program evaluates the data initializer you write whenever it allocates the data object. Its value is converted to the type of the data object and stored in it. If you write no data initializer for an object with dynamic lifetime, its stored value is not valid until your program stores a value in the data object.

If a data object does not have dynamic lifetime, it must have static lifetime. When you write a data initializer for one of its declarations, that is its defining instance. The compiler determines its value from its data initializer at compile time, and stores its value before program startup. A data object with static lifetime has either no linkage, internal linkage, or external linkage. The

compiler creates a defining instance, within the compilation, for such a data object with no linkage or with internal linkage that has no defining instance. Every such declaration is therefore a tentative definition. If the compiler creates a defining instance for a tentative definition, it provides a zero data initializer for each of the components of the data object.

The compiler similarly creates a defining instance for a data object with external linkage, within the compilation, if you write at least one tentative definition for it. You write such a tentative definition at file level by writing no storage class in the declaration. *Therefore, you must write either a data initializer or a tentative definition for every data object you make use of that has external linkage and is not defined in the C library.* For example:

```
int a = 3;          /* defined in this compilation */
int b;              /* defined in this compilation */
extern int c; /* not defined */
static int d; /* defined in this compilation */
```

A compilation that does not provide a defining instance of a data object may declare it with an incomplete type. For example,

```
extern int a[];
```

may be used to access elements of the array **a**.

Data Initializers

You initialize a data object of "scalar" type by writing an equal sign followed by an expression. You may enclose the expression in braces. A scalar type is an integer type, a floating type, or a pointer type. The type of the expression must be "assignment compatible" with the type of the data object. Chapter 5, "Expressions," describes assignment compatibility in the section "Comparing Types". The examples that follow show how you initialize scalar data objects:

```
char space = ' ';
int x = 3;
double d = 2.7;
char *message = "bad file";  /* string becomes char * */
unsigned char *puc = {(unsigned char *)&space};
```

You initialize a data object of array type or structure type, or an *array of unknown content*, by writing an equal sign followed by an expression, or by a list of "data items" you enclose in braces. Each data item is an expression, or a sublist of data items you enclose in braces. If you are initializing a scalar component, a data item must be a constant expression. You separate the data items in a list with commas. You may also write a comma at the end of the list. For example:

```
int vals[4] = {1, 2, 3, 4,}; /* last comma is permitted */
```

You initialize a data object of array type, or an *array of unknown content*, by writing one data item for each of the elements of the array. If you write fewer data items than there are elements, the

compiler provides a zero data initializer for each of the remaining elements of the array. If you write more data items than there are elements, the compiler emits an error message. If the declarator has type *array of unknown content*, the compiler determines the size of the array from the number of data items you provide. It is then no longer an *array of unknown content* and its type is a data object type. You can initialize a data object of type *array of char*, or an *array of unknown content* of the same type, by writing a string constant for the data item, as in:

```
char ab[4] = "ab";       /* 'a', 'b', 0, 0 */
char def[] = "def";      /* 'd', 'e', 'f', 0 */
char ghij[4] = "ghij";   /* 'g', 'h', 'i', 'j' */
char klmn[4] = "klmno"; /* ERROR: too many characters */
```

The terminating null character counts as part of the string constant, but the compiler omits it if there is no room for it in the data object.

You initialize a data object of union type by writing one data item. The data item initializes the first structure member of the union. You initialize a data object of struct type by writing one data item for each structure member. If you write fewer data items than there are structure members, the compiler provides a zero data initializer for each of the remaining structure members of the struct. If you write more data items than there are structure members, the compiler emits an error message. For example:

```
struct Complex zero = {0};
struct Complex one = 1.0;
int identity[3][3] = {{1, 0, 0},
                      {0, 1, 0},
                      {0, 0, 1}};
```

If you leave out any of the inner braces in a data initializer, the compiler determines where they should go. In simple cases, this notation may be a convenient abbreviation. For example, you can write the array **identity** in the example above as:

```
int identity[3][3] = {1, 0, 0, 0, 1, 0, 0, 0, 1};
```

For complex nested structures, however, the compiler may interpret what you intend in a way that you do not expect. You may also find it hard to read a data initializer with braces left out. Do not leave out braces in a complex data initializer.

The compiler restricts the expressions you can write for some data initializers. If the data object has static lifetime, then all expressions must be "constant expressions." Chapter 5, "Expressions," describes these in the section "Constant Expressions." If you initialize an array type with dynamic lifetime, then all expressions must be constant expressions. If you initialize a structure type with dynamic lifetime by writing a list of data items, then all expressions in the data items must be constant expressions. Some examples are:

```
struct Complex frc();
int f() {
     struct Complex one = {1, 0}; /* must be constants */
     struct Complex cx = frc();   /* or a single expression */
     int fibs[] = {1, 1, 2, 3};
```

Type Definitions

You write a type definition by writing a declaration with the storage class **typedef**. Each declarator defines its identifier as a type definition of the type it specifies. Do not write any definitions with any of the declarators. The type definition can subsequently be used wherever a type specifier is permitted. It is not a new type, but a synonym for the type it defines. You use type definitions to summarize complex types in one place, or to introduce mnemonic names for types used in special ways within your source text. The type definition may have *any* type. Some examples are:

```
typedef int I, /* I is int */
     *Pi,      /* Pi is pointer to int */
     *Fri();   /* Fri is function returning pointer to int */
static I i,         /* i is int */
     *pi;      /* pi is pointer to int */
Pi pi;         /* pi is still pointer to int */
typedef Pi *Ppi;   /* Ppi is pointer to pointer to int */
```

You may not use a type definition of function type to declare an identifier alone and then define it. For example:

```
typedef long Frlo(long a);
extern Frlo extfn; /* not a definition */
Frlo intfn
     {.....}       /* ERROR: disallowed */
long extfn(long x)
     {.....}       /* permissible */
```

When you define a function, write its argument list in a function type attribute in its declarator.

Type Names

You write a type name in several contexts where all you need to specify is a type. You write a type cast operator, for example, as a type name in parentheses, as in **(char)** or **(unsigned char *)**. You also use type names to declare argument types in a function prototype. You can also write a type name in parentheses as the argument to the operator **sizeof**.

You write a type name by first writing a normal declaration with an identifier. Do not specify a storage class. Specify a base type and only one declarator. Do not specify a definition. Some examples are:

```
char x;         /* char */
double *pd;          /* pointer to double */
int (*pfi)(); /* pointer to function returning int */
```

There should be no redundant parentheses around the identifier, as in:

```
char *(p);     /* WILL CAUSE TROUBLE (see below) */
```

Drop the identifier and you create a type name. The examples above become:

```
char
double *
int (*)()
char *()  /* TROUBLE: type has changed */
```

To read a type name, determine where the identifier belongs. It will be inside the innermost parentheses that are not empty and are not function prototype argument lists. It belongs to the right of the base type, all asterisks, and the type qualifiers **const** and **volatile**. It is to the left of everything else.

Classification of Types

Here is a summary of all the types in C. They are presented in outline form, grouped by similar properties or similar usage. Learn the names of all the groups in this list:

function types
- *function returning void* types
- *function returning structure* types
- *function returning scalar* types

incomplete types
- *void* type
- *structure of unknown content* types
- *enum of unknown content* types
- *array of unknown content* types
- const incomplete types
- volatile incomplete types

data object types
- array types
- structure types
  - *struct* types
  - *union* types
- scalar types
  - pointer types
    - function pointer types
    - incomplete pointer types
      - *pointer to void* type
      - *pointer to type of unknown content* types
    - data object pointer types

arithmetic types
    floating types
        float type
        double type
        long double type

    integer types
        char type
        signed char type
        unsigned char type
        short type
        unsigned short type
        int type
        unsigned int type
        long type
        unsigned long type
        bitfield types
        signed bitfield types
        unsigned bitfield types
        enum types

const data object types
volatile data object types

The "const incomplete types" and "const data object types" are also referred to collectively as "const types". Similarly, the "volatile incomplete types" and "volatile data object types" are also referred to collectively as "volatile types" and the structure types and enum types are referred to collectively as "composite types."

Chapter 5: Expressions

You use expressions to call a function, compute a value, or alter a data object. Some examples of expressions are:

```
printf("hello\n")        /* call a function */
'a' + 1              /* next value after 'a' */
x = y + z       /* assign value of (y + z) to x */
y = cos(x)           /* assign function value to y */
```

You build expressions from operators, which specify the action you want to perform, and operands, upon which the operators act. An operand can be a constant, such as **3**, the name of a data object, such as **abc**, or a subexpression. The definition of expression is recursive, so a subexpression can, in general, be any expression.

There are many important aspects to expressions in C. You need to know:

* How operators group, in the absence of parentheses
* How the compiler regroups expressions
* Which operators cause side effects, and the extent to which you can control order of subexpression evaluation
* How your program converts between arithmetic types
* How your program converts between pointer and integer types
* What type conversions the compiler permits on assignment
* The function of each of the operators
* What restrictions apply to "compile time" or "constant" expressions

This chapter describes each of these aspects.

Grouping and Precedence

You use parentheses to tell the compiler exactly which operands you want to associate with which operators. For example, the expression **a*(2+c)** calls for 2 to be added to **c** (via the operator **+**), and the result multiplied by **a** (via the ***)**. If you write this expression without parentheses, as in **a*2+c**, the compiler must apply a series of rules to determine how to group the operands with the operators.

The various operators have different precedence. The rules for precedence determine how tightly operands bind to their operators. Multiplication, for example, has higher precedence than addition, so the compiler groups the expression `a*2+c` as `(a*2)+c`.

When operators have the same precedence, the order in which the compiler groups them is either *left to right* or *right to left*. For example, the compiler groups the expression `x/y/z` as `(x/y)/z`, because the divide operator `/` groups left to right. On the other hand, the compiler groups `x |= y |= z` as `x |= (y |= z)` since the assignment operators group right to left.

In order of descending precedence, the operators are:

1. The addressing operators: function call `x(y)`, subscript `x[y]`, point at member `x->y`, and select member `x.y`. All group right to left.

2. The unary operators: logical NOT `!x`, bitwise NOT `~x`, preincrement `++x`, postincrement `x++`, predecrement `--x`, postdecrement `x--`, plus `+x`, minus `-x`, indirect on `*x`, address of `&x`, size of `sizeof x`, and type casts such as `(double)x`. All group right to left.

3. The multiplicative operators: multiply `x*y`, divide `x/y`, and remainder `x%y`. All group left to right.

4. The additive operators: add `x+y` and subtract `x-y`. Both group left to right.

5. The bitwise shift operators: left shift `x<<y` and right shift `x>>y`. Both group left to right.

6. The relational operators: less than `x<y`, less than or equal `x<=y`, greater than `x>y`, and greater than or equal `x>=y`. All group left to right.

7. The equality operators: equal to `x==y` and not equal to `x!=y`. Both group left to right.

8. The operator bitwise AND `x&y` groups left to right.

9. The operator bitwise exclusive OR `x^y` groups left to right.

10. The operator bitwise inclusive OR `x|y` groups left to right.

11. The operator logical AND `x&&y` groups left to right.

12. The operator logical OR `x||y` groups left to right.

13. The operator conditional `x?y:z` groups right to left. This is a "ternary" operator, because its "right" operand is a pair of alternative expressions separated by a colon. An example is

    ```
    a<0 ? -1 : 1
    ```

    which has the value `-1` if `a<0` is true, or the value 1 if `a<0` is false.

14. The assignment operators: gets `x=y`, gets multiplied `x*=y`, gets divided `x/=y`, gets remainder `x%=y`, gets added `x+=y`, gets subtracted `x-=y`, gets left shifted `x<<=y`, gets right

shifted **x>>=y**, gets AND **x&=y**, gets exclusive OR **x^=y**, and gets inclusive OR **x|=y**. All group right to left.

15. The operator comma **x,y** groups left to right.

As an example, the more complex expression

```
a = b = x & y | z
```

groups the same as

```
a = (b = ((x & y) | z))
```

Always use parentheses to remove ambiguity when working with operators you use infrequently. Otherwise, you may misread the expression later.

Regrouping

The C compiler performs as much expression evaluation at compile time as it can. This capability permits you to write complex expressions in places where the compiler must know the value at compile time. The section "Constant Expressions," later in this chapter, discusses compile time expressions. Compile time evaluation can also significantly reduce the amount of computation your program actually performs every time you run it. In looking for subexpressions that it can compute at compile time, the compiler can regroup expressions in certain ways. It may perform regrouping even if you write parentheses to show how you intend to group the operands.

As an example, the compiler can regroup the expression

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

in a number of ways, including

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

even though the add operator ordinarily groups left to right, and even though your explicit parentheses suggest a different grouping.

The compiler can also regroup expressions that use any of the "commutative" operators (*, +, &, ^, and |). An operator is commutative if the order in which you apply it to multiple operands does not change the result. For example, **a&b&c** yields the same result whether you write it as **(a&b)&c** or as **a&(b&c)**.

If you want to enforce a given grouping within an expression, you can use the plus operator **+x**, as in

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

Using the plus operator this way defeats the regrouping and ensures that the compiler will add **c** to **d** before combining the result with the other operands.

Side Effects and Order of Evaluation

A "side effect" is a change in the value stored in a data object, or a change in the state of a file, as a byproduct of evaluating an expression. Function calls, the increment and decrement operators, and the assignment operators all cause side effects. Your program evaluates some expressions, known as "void expressions," only for their side effects. A void expression either has no value, because its type is *void*, or your program discards its value.

The following expressions all have side effects:

```
printf("hello world\n")
++a
a = 1
```

If an expression has more than one side effect, the order in which your program evaluates any subexpressions can yield unexpected results. In many cases, however, the C language does not restrict order of evaluation enough to make the evaluation of such expressions completely predictable. For example, your program can evaluate the arguments of the function call

```
f(++a, ++a)
```

in arbitrary order, so you have no way of precisely determining the values of the arguments. As a general rule, avoid writing expressions with more than one (related) side effect.

Certain operators do provide a predictable order of evaluation of their operands. You can use these operators to write robust expressions, including expressions with multiple side effects. The operators logical AND **&&** and logical OR **||**, for example, ensure strict left to right evaluation. You can therefore write

```
0 <= i && i < sizeof a && a[i] != 'c'
```

and be sure that **a[i]** will never be outside the *array of char* data object **a**. Similarly, the comma operator **x,y** evaluates its **x** operand completely before evaluating its **y** operand. So

```
C1 = getchar(), C2 = getchar()
```

stores the first input character in **C1** and the second one in **C2**.

The compiler evaluates the arguments to a function call before calling the function. Also, an assignment operator causes the compiler to modify storage before using the assigned value further in an expression. The descriptions of the operators that follow state any restrictions on order of evaluation for a given operator.

Types and Classes of Expressions

Two important attributes of every expression are the type of its result and its "class." These are determined as follows:

Types of Expressions

Every expression or subexpression has a type. If the expression is an identifier, you determine its type when you declare it. If an expression is a constant, the rules for typing constants (described in Chapter 2, "Elements of the C Language") determine its type. If you enclose an expression in parentheses, its type is that of the expression in parentheses. Otherwise, the expression must be an operator with subexpressions for operands. Each operator has its own rules for determining its type, based on the types of its operands. If you start with the simplest subexpressions, you can determine their types, then work up through the operators until you determine the type of the expression as a whole.

For example, the expression:

```
2.4 + (3L - 2)
```

has the constant operands **2.4** with type *double*, **3L** with type *long*, and **2** with type *int*. Subtracting a value of type *int* from a value of type *long* gives a result of type *long*. Adding a value of type *long* to a value of type *double* gives a result of type *double*. So the type of the expression is the type of the operator **+**, which is type *double* in this case.

Classes of Expressions

Every expression or subexpression is in one of four classes:

* A "void expression," which yields no value
* A "function designator," which designates a function of some type
* An "lvalue," which designates a data object of some type
* An "rvalue," which yields a value of some data object type.

Just as with type, you can determine the class of an expression by starting with the simplest subexpressions and working up through the operators. Enclosing an expression in parentheses does not change its class.

A void expression is one whose type is *void*. As discussed earlier, the compiler evaluates such expressions for their side effects only. An example is the C library function

```
srand(0)
```

which provides an initial value for a random number generator. **srand** has type *function returning void*. If any other class of expression occurs where the compiler expects a void expression, the compiler evaluates it and discards the result. The function **printf**, for example, returns a value every time your program calls it, but the expression that calls it often discards that value.

A function designator is one whose type is *function returning T*, where *T* is a valid type. In the example above, **srand** is a function designator, which the compiler uses to determine which function to call. There is no context where the C language requires a function designator, so there are no rules for converting

other classes of expression to this class.

An lvalue can also be called a "data object designator expression." It designates the location of a data object for the purpose of taking its address, obtaining its stored value, or altering its stored value. An lvalue must have a data object type. An identifier you declare to be a data object is an lvalue. Expressions such as **p->m** and **a[i]** are always lvalues. ***p** is an lvalue if its type is a data object type. **x.m** is an lvalue if **x** is an lvalue. No other class of expression may occur where the compiler requires an lvalue.

If you want to use an lvalue to alter the value stored in a data object, it must be a "modifiable lvalue." This means it must not be of type *array of T*, and it must not be a const type.

Some examples of lvalues are:

```
&x + 1              /&x is an lvalue */
abc = 0             /* abc is a modifiable lvalue */
p->last_char = '3'  /* p->last_char is modifiable lvalue */
*f() = 27           /* *f() is a modifiable lvalue */
```

Some examples that are *not* lvalues are:

```
&frs().abc      /* ERROR: frs().abc is NOT an lvalue */
x + 1 = 3       /* ERROR: x + 1 is NOT a modifiable lvalue */
```

An rvalue is an expression that is not in any of the other three classes. It has a data object type other than *array of T*. It also has a value. A void expression may not occur where the compiler requires an rvalue. If a function designator, of type *function returning T*, occurs where the compiler requires an rvalue, its value is the address of the function and its type is *pointer to function returning T*. So if **f** is a function designator, writing

```
g(f)
```

is the same as writing

```
g(&f)
```

since the argument to a function must be an rvalue.

If an lvalue occurs where the compiler requires an rvalue, and its type is not *array of T*, then its value is the value stored in the data object you designate by the lvalue and its type is the type of the lvalue. So if **x** is of type *int*, writing

```
g(x)
```

uses the current value in **x** as the value of the argument. The function **g** cannot modify the value stored in **x** by modifying the value you pass for the argument.

If an lvalue is of type *array of T*, then its value is the *address* of the first element of the array (having subscript zero) and its type is *T*. So if **a** is an array, writing

```
g(a)
```

is the same as writing

```
g(&a[0])
```

This last rule lets you use the name of an array in contexts where you need a pointer into the array. While this is generally convenient, it often leads to confusion between array types and pointer types.

Arithmetic Conversions

Many of the arithmetic operators follow the same rules for determining their result type from the types of their operands. The first rule is that the *wider* of the two operand types determines the type of the result. If that type is not at least as wide as *int*, then the compiler converts the type of the result to *int*. Another rule requires that your program must "widen" each of the operands to the result type before the operation takes place. As an example, if you add the *short int* **sh** to the *double* **d**, as in **sh+d**, then your program must convert **sh** to a *double* with the same numeric value as **sh** before adding it to **d**. The result is of type *double*.

Widening Conversions

The compiler determines the width of a type in part by how the target machine represents it. In general, the wider the type, the greater the range of values it can correctly represent. On System/370, the compiler represents the type *int* as a 32 bit two's complement integer. Types narrower than *int* include *signed char*, *char*, *unsigned char*, *short*, *unsigned short*, and *signed bitfield*. The compiler may represent type *enum* as narrower than *int*. The plain type *char* has the same representation as *unsigned char*. An *unsigned bitfield* of fewer than 32 bits is also narrower than *int*. When you see the phrase "the compiler widens to *int*," in the descriptions of operators later in this chapter, it means the following:

> The compiler converts all arithmetic types narrower than *int* to an *int* that has the same value as the original type. An *unsigned bitfield* of 32 bits widens to *unsigned int*. On System/370, a plain *bitfield* of 32 bits also widens to *unsigned int*. Any other type is left alone. If the operand is not a constant expression, then the compiler produces executable code to make your program perform the proper conversion at runtime.

Although *long* has the same representation as *int* on System/370, the compiler considers *long* to be a wider type. The compiler considers *long* to be wider than *unsigned int*, even though *long* cannot represent all of the values of type *unsigned int* on System/370 and other computers. The compiler represents the type *float* as a 32 bit floating point data item. It considers *float* to be wider than any of the integer types even though it cannot retain 32 significant bits. The type *long double* has the same representation as *double*: a 64 bit floating point data item. The widening order of the types that represent arithmetic results is:

int, *unsigned int*, *long*, *unsigned long*, *float*, *double*, and *long double*.

The following examples show how the compiler applies the arithmetic conversion rules to addition of many of the arithmetic types. In all the cases shown **ch** is of type *char*, **in** is of type *int*, **ui** is of type *unsigned int*, **lo** is of type *long*, **ul** is of type *unsigned long*, **fl** is of type *float*, and **do** is of type *double*:

```
ch + ch   /* widen both to int, result is int */
ch + in   /* widen ch to int, result is int */
ch + ui   /* widen ch to unsigned int, result is unsigned int */
ch + lo   /* widen ch to long, result is long */
ch + ul   /* widen ch to unsigned long, result is unsigned long */
ch + fl   /* widen ch to float, result is float */
ch + do   /* widen ch to double, result is double */

in + in   /* result is int */
in + ui   /* widen in to unsigned int, result is unsigned int */
in + lo   /* widen in to long, result is long */
in + ul   /* widen in to unsigned long, result is unsigned long */
in + fl   /* widen in to float, result is float */
in + do   /* widen in to double, result is double */

ui + ui   /* result is unsigned int */
ui + lo   /* widen ui to long, result is long */
ui + ul   /* widen ui to unsigned long, result is unsigned long */
ui + fl   /* widen ui to float, result is float */
ui + do   /* widen ui to double, result is double */

lo + lo   /* result is long */
lo + ul   /* widen lo to unsigned long, result is unsigned long */
lo + fl   /* widen lo to float, result is float */
lo + do   /* widen lo to double, result is double */

ul + ul   /* result is unsigned long */
ul + fl   /* widen ul to float, result is float */
ul + do   /* widen ul to double, result is double */

fl + fl   /* result is float */
fl + do   /* widen fl to double, result is double */

do + do   /* result is double */
```

Converting an *int* to a *long*, or an *unsigned int* to an *unsigned long*, requires no change of representation or value. Converting an *int* or *long* to an *unsigned int* or an *unsigned long* requires no change of representation. Your program reinterprets negative values as very large positive values. Converting any of the integer types to a *float* may result in loss of significance. Your program truncates the value toward zero. The conversion of any of the integer types or a *float* to a *double* or a *long double* is exact.

Unsignedness Preserving Rules

Using a compiler option, you can instruct the C compiler to enforce an alternate set of rules for widening operands. These rules are called the "unsignedness preserving rules," as opposed to the "value preserving rules" adopted for the proposed ANSI standard. The *C Compiler User's Guide* for your system describes how to enforce this alternate set of rules.

Under the unsignedness preserving rules, the types *unsigned char*, *unsigned short*, all plain *bitfields*, and all *unsigned bitfields* widen to *unsigned int*.

If you write an expression whose operands are *unsigned int* and *long*, the resulting type is *unsigned long*. All other rules are the same.

You use this option to compile large programs written for a compiler that enforces unsignedness preserving rules, until you can locate and correct any expressions whose meanings have changed significantly. On a two's complement machine that ignores integer overflow such as System/370, few such cases arise. Look for *unsigned char* and *unsigned short* operands combined with negative integers in expressions such as **x<y**, **x<=y**, **x>y**, **x>=y**, **x>>y**, **x/y**, and **x%y**. Write type casts whenever you wish to enforce a given result type. For example,

```
unsigned char uc;
if (uc-'0'<='9')    /* change to (unsigned)(uc-'0')<='9' */
```

Narrowing Conversions

If you write an expression that assigns a narrower type to a wider type, then your program widens the value stored in the data object in the way just described. If you write an expression that assigns a value of the same type as the data object, then your program stores the value unchanged. If you write an expression that assigns a wider type to a narrower type, then your program must "narrow" the value stored in the data object by applying one or more of the following conversion rules.

Your program converts the types *long double* and *double* to *float* by discarding excess significance. It truncates the value toward zero. It converts all floating types to integer types by discarding all fraction bits and truncating the value toward zero. An overflow occurs if the value is too large for the computer to represent in the integer type you specify, or if the value is negative and the type you specify is an unsigned integer type. On System/370, your program retains the low order bits that it can represent in the specified type. This is an unreported error.

Similarly, if your program must convert an integer type to an integer type having fewer bits in its representation, the value is unchanged wherever possible. If the value is negative and the specified type is an unsigned integer type, then your program retains the low order bits that it can represent in the specified type. If the value is too large for the computer to represent in the specified integer type, an overflow occurs. On System/370,

your program stores the low order bits that it can represent in the specified type. This is an unreported error if the specified type is not an unsigned integer type.

The following examples show some of the arithmetic conversion rules that the compiler applies to narrowing assignments. In all the cases shown, **sc** is of type *signed char*, and **uc** is of type *unsigned char*:

```
sc = 5          /* exact */
sc = -1         /* exact */
uc = 5          /* exact */
uc = -1         /* valid, value stored is 255 */

sc = 25.4       /* value stored is 25 */
sc = -25.4      /* value stored is -25 */
uc = 25.4       /* value stored is 25 */
uc = -25.4      /* ERROR: value stored is 231 */
```

Unsigned Conversions

Unsigned integer arithmetic is more accurately "modulus" arithmetic, sometimes called "wraparound" arithmetic. Overflow cannot occur, because adding one to the largest unsigned integer value wraps around to zero, by definition. Similarly, subtracting one from zero wraps around to the largest unsigned integer value, also by definition. For maximum portability, your program should avoid the conversions described above that result in errors. Unsigned arithmetic has fewer such cases than signed integer arithmetic. You should therefore use unsigned integers in situations where you want your program to ignore loss of significant bits.

Pointer Conversions

You can assign the value of expressions of certain types to a data object of type *pointer to T*. You can also write a type cast operator in front of an expression to convert between pointer types and certain other types. All types of the form *pointer to T* fall into one of the three groups: function pointers, incomplete pointers, and data object pointers. The rules for converting among pointer types in different groups are discussed below.

Converting Integers to Pointers

You may assign an integer expression whose value is zero to any of these pointer types. In all cases, the result is a "null pointer," which compares equal to integer zero. No function or data object in C has an address that compares equal to a null pointer.

You may type cast an integer expression whose value is nonzero to any of these pointer types. On System/370, a type cast does not change the representation of the integer value, so the compiler treats the integer as an absolute machine address. The result of using such a pointer to call a function or to access a data object is *very* machine dependent. A typical application has no valid use

for such pointers. Unless you know precisely what the effect will be, you should avoid type casting any integer other than zero to a pointer type.

Converting Function Pointers

You can convert any function pointer to another function pointer by means of a type cast operator. Do not use the converted value to call a function, because the protocols may differ for functions you declare differently. However, you can convert the value back to its original type and then use the resulting value to call a function. *All function pointers have the same representation, so you may use any kind of function pointer to convey the address of any kind of function.* For example, if **pfi** is of type *pointer to function returning int*, **fi** is of type *function returning int*, and **pfd** is of type *pointer to function returning double*:

```
pfi = &fi      /* types are the same */
pfi()              /* same as fi() */
pfd = (double (*)())pfi /* pfi is cast to type of pfd */
pfd()    /* ERROR: stored value is unsuitable for call */
pfi = (int (*)())pfd    /* pfd is cast to type of pfi */
pfi()              /* stored value is suitable for call */
```

Converting Data Object Pointers

You may convert any data object pointer to another data object pointer by means of a type cast operator. Since you can treat any data object as an *array of char* whose size is the number of bytes in the data object, you can always convert a data object pointer to a *pointer to char*. If you convert a data object pointer to some other data object pointer type, however, and then use the converted value to access the data object, the results are unpredictable. The results are also unpredictable if you convert the pointer value back to its original type and then use the resulting pointer value to access the data object. Do this only where the intermediate type points to data objects that are no more strictly aligned on storage boundaries than data objects pointed to by the original type. Since *pointer to char* always has the least strict alignment, it is the best intermediate type to use for conveying data object pointers if you want to write a portable program. System/370 favors certain storage alignment restrictions for improved performance, but permits any alignment for any data object.

As an example, if **pi** is of type *pointer to int*, **i** is of type *int*, and **pc** is of type *pointer to char*:

```
pi = &i        /* types are the same */
*pi = 3        /* same as i = 3 */
pc = (char *)pi    /* pi is cast to type of pc */
*pc = 'a'      /* part of i overwritten */
pi = (int *)pc /* pc is cast to type of pi */
*pi = 2        /* same as i = 2 */
```

Converting Incomplete Pointers

You can assign or type cast an incomplete pointer to a data object pointer, and you can assign or type cast a data object pointer to an incomplete pointer. A *pointer to void* has the same representation as *pointer to char*, so it is just as useful for conveying data object pointers. A *pointer to void* has the added advantage that the compiler converts it automatically on assignment, without an explicit type cast. You can therefore use it as the type of a function call argument or return value that can convey any data object pointer. This minimizes the need for writing type casts when calling the function. You cannot use an incomplete pointer to access a data object. You can, however, use such a type to convey data object pointer values.

Converting Pointers to Integers

There is no provision for directly type casting a function pointer to a data object pointer, a function pointer to an incomplete pointer, a data object pointer to a function pointer, or an incomplete pointer to a function pointer. The C language considers function pointers, integers, and the remaining pointer types to have as many as three different representations. Since System/370 represents all pointers the same as values of type *int*, you can type cast any pointer to *int*, then to any other pointer type without loss of information, although this is not recommended. Observe the exceptions described above, however, if you want your program to be portable.

Pointer Arithmetic

You can add an integer to a data object pointer. The result is of the same pointer type, and you can use the value to designate a data object different from the original pointer value. Adding one to a pointer actually increments the pointer value by the number of bytes the compiler uses to represent a data object of the type pointed to. So if **p** points to a member of an array, **p+1** points to the next member of that array, if there is one. If there is a next member, ***(p+1)** is an lvalue that designates that next member. If not, then the expression **p+1** is valid, but ***(p+1)** is not. You may often want to increment a pointer until it points just past the end of an array, as part of a control loop that processes all the members of the array.

You can also subtract an integer from a data object pointer. As with addition, the compiler multiplies the integer value by the number of bytes it uses to represent a data object of the type pointed to. So if **p** points to a member of an array, **p–1** points to the immediately preceding member of that array, if there is one. If there is an immediately preceding member, ***(p–1)** is an lvalue that designates that preceding member. If not, then the expression **p–1** is *not* valid, and neither is ***(p–1)**. Thus, you should never decrement a pointer until it points before the beginning of an array.

Any pointer value you generate by adding or subtracting an integer must point inside the same data object as the original pointer or just beyond it, as described above. If not, then the expression is not valid. If you compare an invalid pointer value to other pointer values, you may get unpredictable results. Even subtracting the value you just added may not result in a valid pointer.

Every data object in C is composed of an integral number of contiguous bytes of storage. You can treat every data object as if it were a union, one of whose members is of type *array of char* with a size equal to the number of bytes the compiler uses to represent the data object. The address of element zero of the array is the same as the address of the data object. You can therefore copy a data object, read it in, or write it out character by character, by advancing a *pointer to char* through it from beginning to end. These actions are commonplace in C.

Comparing Types

The compiler compares two types in one of two ways: either they must be "the same" or they must be "assignment compatible."

Same Types

The compiler checks that two types are the same when you redeclare an identifier. Two pointers are assignment compatible if they point to the same type. Whether you use a type definition to summarize part of the type information for one type does not affect whether two types are the same. The rules for determining if two types are the same are discussed below.

Two types are the same if they are identical. The type *array of T* of unknown content is the same as the type *array of T* with known content, if their types *T* are the same. The type *function returning T* with unspecified arguments is the same as the type *function returning T* with specified arguments (a function prototype), if their types *T* are the same. If both have specified arguments, the number and corresponding types of the arguments must be the same.

Two structures that you declare within one compilation with no tags or with different tags are never the same, even if they declare fields of the same type, with identical names, in identical order. Two structures that you declare in different compilations are the same, however, if they declare fields of the same type in identical order.

Some examples are:

```
int a[];
int a[5];       /* same type */

typedef char *Pc;
char *pc;
Pc pc;                  /* same type */
```

```
int f(int x, int y);
int f(int a, int b);    /* same type */
int f();              /* still same type */
```

Assignment Compatibility

When you write an expression that assigns the value of an expression to a data object, the types of the data object and expression value must be assignment compatible. When you call a function, the compiler effectively assigns each of the argument expressions to the corresponding actual argument you use within the called function. If you specify the argument types by writing function prototypes, the compiler checks each actual argument with its corresponding function prototype argument for assignment compatibility. When you write a type cast operator in front of an expression, you specify that the operand value be converted from the operand type to the type specified in the type cast. If the two types are scalar and assignment compatible, the compiler always permits the type cast.

The rules for determining whether a value of some type is assignment compatible with some other type are as follows:

Assignment is defined only for storing in data object types other than array types and const types. The C language does not permit assignment of a pointer to any const type to a pointer to a nonconst type. The compiler does not permit assignment of a pointer to any volatile type to a pointer to a nonvolatile type. The type qualifiers **const** and **volatile** are otherwise ignored.

If the two types are the same, they are assignment compatible. If both the types in question are arithmetic types, they are assignment compatible. The rules for converting values among arithmetic types are presented earlier in this chapter.

You may assign a value of the type *pointer to T1* to a data object of type *pointer to T2*, if their types *T1* and *T2* are the same. You may assign an integer expression whose value is zero to any pointer type. You may assign a value of type *pointer to void* to any data object pointer or incomplete pointer. You may assign the value of a data object pointer or an incomplete pointer to a data object of type *pointer to void*.

Some examples of assignment compatibility are:

```
int i = 3;          /* same type */
int j = 3.5;   /* both arithmetic */

int *pi = &j; /* same type */
void *pv = &j; /* void * is compatible */
```

If you want to write an assignment of a form not covered by these rules, you can generally write a type cast to convert the expression to a type that is assignment compatible with the data object you wish to modify. You can convert any scalar type to any other scalar type by writing at most two type casts. The conversion may, however, not be portable.

The C Operators

C offers an exceptionally powerful set of operators. This section describes each operator, how you write it, and what it does.

Addressing Operators

You can select a portion of a data object in one of several ways, by subscripting, as with **x[y]**, by pointing at a structure member with a structure pointer, as with **x->y**, or by selecting a structure member from a structure, as with **x.y**. Here and in the discussion below, the symbols x and y stand for the two operands. Addressing operators and function calls are at the same level of precedence. They group right to left.

Subscript Operator

The operator "subscript" **x[y]** uses the values of its operands to produce an lvalue that designates an element of an array. Both operands must be rvalues. One of the operands must be a data object pointer, the other must be an integer type. You usually write the data object pointer to the left (in place of **x**) and the integer operand inside the square brackets (in place of **y**). This notation is similar to that in several other programming languages, where **x[y]** designates the **y**th element of the array designated by **x**. If the data object pointer is of type *pointer to T*, the result is of type *T*. The expression **x[y]** is entirely equivalent to the expression ***(x+(y))**.

When you write an expression such as **a[3]**, to select element 3 from the *array of int* **a**, several actions occur. First, the compiler rewrites the expression as ***(a+(3))**. Since **a** is an lvalue and the add operator requires rvalue operands, the compiler must convert **a** to an rvalue. An lvalue of type *array of int* becomes a data object pointer of type *pointer to int* that points at element zero of the array. This is true even if **a** is an *array of unknown content*. Adding an integer 3 to a *pointer to int* requires that the 3 first be multiplied by **sizeof (int)**. On System/370, this operation produces 12, the byte offset of element 3 of the array. Finally, the operator indirect on ***** uses the value of its data object pointer operand to produce an lvalue. That lvalue designates element 3 of the array. You can use the lvalue to store a new value in the array element, as in **a[3]=2**, or to obtain its value, as in **a[3]+2**, or to obtain its address, as in **&a[3]**. The subscript operator expresses a selection process used frequently in C programs.

If you have an array with multiple dimensions, you simply repeat the subscripting process. For example, if **b** is of type *array of array of int* and **i** is of type *int*, the expression **b[i]** is an lvalue of type *array of int*. You can subscript it further, as in **b[i][7]**, which is an lvalue of type *int*. There is no limit to the number of dimensions you can specify for an array, so there is no limit to the number of subscripts you can write in an expression. You must be careful, however, not to write an expression such as **b[i,7]**. This is *not* equivalent to **b[i][7]**. The comma operator **,** causes the compiler to evaluate **i** as a void expression, so its

value is discarded. The resulting subscript is the *right* operand of the comma operator, giving an expression equivalent to **b[7]**. The only way to write multiple subscripts is to enclose each in its own square brackets.

Point at Member Operator

The operator "point at member" **x->y** produces an lvalue that designates the structure member whose name is **y** in the structure pointed at by **x**. The operand **x** must be an rvalue. It must be a data object pointer of some structure type. Declare the operand **y** previously as the name of a structure member within the designated structure. The value your program generates to designate the structure member **y** is the value of the data object pointer **x** plus the offset in bytes of structure member **y** from the start of the structure. This offset is not multiplied by any size factor, unlike when you add an integer operand to a data object pointer. The type of the result is the type of the structure member **y**.

Select Member Operator

The operator "select member" **x.y** obtains the structure member whose name is **y** in the structure **x**. If **x** is an lvalue, **x.y** is an lvalue that designates the structure member **y**. Otherwise, **x.y** is an rvalue whose value is the value of structure member **y**. The operand **x** must be a structure type, and you must declare **y** previously as the name of a structure member within the designated structure. If **x** is an lvalue, the value your program generates to designate the structure member **y** is the address of **x** plus the offset in bytes of structure member **y** from the start of the structure. As with **x->y**, this offset is not multiplied by any size factor. The type of the result is the type of the structure member **y**.

If **x->y** is a valid expression, then **(*x).y** is always valid. If **x.y** is a valid expression, and if **&x** is also valid, then **(&x)->y** is always valid.

You can write an rvalue having a structure type in several ways. For example, if **frs** designates a function that returns a structure, then **frs()** is such an rvalue. If **m** is a structure member of that structure, then **frs().m** is the value of structure member **m** returned by the function call **frs()**. You cannot write **&frs().m**, because the expression **frs().m** does not designate a data object. Since it is an rvalue, it only has a type and a value.

Function Calls

The operator "function call" **x(y)** calls the function you designate by **x**, passing it the argument list **y**. If **x** is not a function designator, of type *function returning T*, then it must be a function pointer, of type *pointer to function returning T*, or an identifier that has no declaration in scope. You can use a function pointer **x** to call a function by writing either **x(y)** or **(*x)(y)**. When you use an undeclared identifier to designate a

function, the compiler implicitly declares the identifier within the current block as if you had written:

```
int x();
```

right after the left brace that begins the current block. When you call a function, your program remembers where it left off in evaluating the expression by saving its "calling environment," and passes control to the first statement of the function. When the function returns control by executing a *return* statement, your program restores its calling environment and continues evaluating the expression where it left off. If *T* is *void*, or if the *return* statement executed within the function specifies no return value, evaluate the function call as a void expression. Otherwise, the function returns a value of type *T*, and the function call is an rvalue. A function may return structure types as well as scalar types.

The argument list **y** may be empty, as in **x()**, or a single expression, as in **x(arg)**, or a list of expressions you separate with commas, as in **x(arg1, arg2, arg3)**. If the argument list is empty, you must still write the empty parentheses to specify a function call.

A function call allocates a data object for each of the expressions in the argument list. Each of the argument expressions is an rvalue whose value is stored in the corresponding argument data object. Arguments may have structure types as well as scalar types. The function may alter the values stored in any of its argument data objects. When the function returns, the compiler deallocates the argument data objects for this function call. A function may call itself recursively, or call other functions that call the original function in turn. Each call allocates and frees a different stored calling environment and a different set of argument data objects, just as the function itself allocates and frees different sets of automatic data objects. Each execution of a *return* statement within the function returns control to the last expression that called the function.

When you write the declaration for **x**, you can choose to declare its argument types. This form of declaration is a function prototype. It is described in Chapter 4, "Declarations." Some examples of function prototypes are:

```
void f(void);       /* no arguments permitted */
void g(double);          /* one double argument */
void h(char *, int);     /* two arguments */
void j(char *, ...);     /* one or more arguments */
```

If you choose this form, then the argument list in the function call must have the number of arguments the declaration requires. The compiler checks that each argument is assignment compatible with the corresponding type in the declaration. The compiler then converts the value of the argument expression to that type before it is stored in the data object. For example, you can use the function prototypes shown above with the following function calls:

```
f()             /* no arguments */
g(3)            /* 3 is converted to 3.0 */
h("hello", 3)  /* "hello" becomes char * as an rvalue */
j("I F I\n", 3, 2.7, 1) /* first argument checked */
```

The following function calls are *not* valid:

```
f('a')          /* ERROR: too many arguments */
g("abc")  /* ERROR: not assignment compatible */
h('a', 3) /* ERROR: first not assignment compatible */
j()       /* ERROR: not enough arguments */
```

When you write the declaration for x, you can also choose *not* to declare some or all of its argument types, as in

```
void j(char *, ...);     /* extra arguments not checked */
void k();              /* no argument information */
void m(a1, a2)         /* definition does not provide */
     int a1, a2;          /* argument information for calls */
     {.....}
```

If that is your choice, then the compiler checks each call to the function in a different way. First it determines the type of each argument rvalue. The compiler widens each argument to *int*, and converts any argument of type *float* to type *double*. From these converted types, the compiler builds an implicit argument declaration, which has the proper number of arguments. The arguments also have types that are assignment compatible with the expressions in the function call argument list. The compiler then allocates argument data objects and stores argument values just as it does in the presence of an explicit declaration for the function arguments. Each such function call gets its own implicit argument declaration. The compiler does not check that different function calls on the same function have implicit argument declarations that agree in number or corresponding types. Nor does it check the function calls against the definition, if you provide the definition earlier in the same compilation. For maximum portability, however, all function calls must have implicit argument declarations that match the function definition.

For example, you may use the declarations shown in the last series of examples above with any of the following function calls:

```
k()             /* no arguments */
k("hello", "world")/* like void k(char *, char *) */
k(3.2f)         /* like void k(double) */
m(1, 2)         /* like void m(int, int) */
```

The following function calls are *not* valid:

```
k(m(1, 2))      /* ERROR: void can't be an rvalue */
m(3)            /* ERROR: wrong number of arguments */
m(3.0, 2)       /* ERROR: won't match definition */
```

The ability to declare function arguments so that the compiler can check them is a recent addition to the C language. Many existing programs rely on the implicit conversion described above to match arguments passed on a function call with those the function definition expects. If you declare all your function arguments, your

program will be less prone to errors while it is being written and later maintained.

OS Calling Sequence

If you wish to call a function that uses the standard OS calling sequence, such as a function written in another language, declare the function in a special way. Chapter 7, "The Preprocessor," describes how to specify that a function uses the OS calling sequence, in the section "Pragmas."

Unary Operators

You write a unary operator either to the left of its operand, as in **~x**, or to the right, as in **x++**. Here, and in the examples below, **x** stands for the operand. Unary operators group right to left.

Logical NOT Operator

The operator "logical NOT" **!x** compares its operand against zero, to produce an rvalue. If the operand compares equal to zero, the value of the result is one. Otherwise the value is zero. The operand must be an rvalue of scalar type. For example, **!3** has the value 0, **!0** has the value 1, and **!2.78** has the value 0. The type of the result is always *int*.

Bitwise NOT Operator

The operator bitwise NOT **~x** inverts each of the bits of its operand value, to produce an rvalue. A one bit becomes zero, a zero bit becomes one. The operand must be an rvalue of integer type, which the compiler widens to *int*. For example, on System/370, **~0** has the value 0xFFFFFFFF (all one bits), and **~0x76543210** has the value 0x89ABCDEF. The type of the result is the type of the widened operand.

Preincrement Operator

The operator "preincrement" **++x** adds one to the value stored in the data object you designate by its operand, to produce an rvalue. The operand must be a modifiable lvalue. It must have arithmetic type or a data object pointer type. The result is the *new* value stored in the data object. For example, if **x** initially holds the value 3, **++x** leaves the value 4 in **x** and yields the value 4. Its type is the type of the operand, which the compiler widens to *int*. The expression **++x** is entirely equivalent to the expression **(x+=1)**, as described later in this chapter.

Predecrement Operator

The operator "predecrement" **--x** subtracts one from the value stored in the data object you designate by its operand, to produce an rvalue. The operand must be a modifiable lvalue. It must have arithmetic type or a data object pointer type. The result is the new value stored in the data object. For example, if **x** initially holds the value 3, **--x** leaves the value 2 in **x** and yields the value 2. Its type is the type of the operand, which the

compiler widens to *int*. The expression **--x** is entirely equivalent to the expression **(x-=1)**, described later in this chapter.

Postincrement Operator

The operator "postincrement" **x++** adds one to the value stored in the data object you designate by its operand, to produce an rvalue. The operand must be a modifiable lvalue. It must have arithmetic type or a data object pointer type. The result is the *old* value stored in the data object. For example, if **x** initially holds the value 3, **x++** leaves the value 4 in **x** and yields the value 3. Its type is the type of the operand, which the compiler widens to *int*.

Postdecrement Operator

The operator "postdecrement" **x--** subtracts one from the value stored in the data object you designate by its operand, to produce an rvalue. The operand must be a modifiable lvalue. It must have arithmetic type or a data object pointer type. The result is the *old* value stored in the data object. For example, if **x** initially holds the value 3, **x--** leaves the value 2 in **x** and yields the value 3. Its type is the type of the operand, which the compiler widens to *int*.

Plus Operator

The operator "plus" **+x** does nothing to the value of its operand, to produce an rvalue. The operand must be an rvalue of arithmetic type, which the compiler widens to *int*. For example, **+3** yields the value 3, and **+18.2** yields the value 18.2. The type of the result is the type of the widened operand. You may also use the plus operator to prevent regrouping, as described earlier in this chapter in the section "Regrouping."

Minus Operator

The operator "minus" **-x** negates the value of its operand, to produce an rvalue. The operand must be an rvalue, of arithmetic type, which the compiler widens to *int*. For example, **-3** has the value -3, and **-18.2** has the value -18.2. The type of the result is the type of the widened operand.

Indirect On Operator

The operator "indirect on" ***x** uses the value of **x** to designate a data object or a function. The operand must be an rvalue, of type *pointer to T*. If the operand is a data object pointer that points to a valid data object, the result is an lvalue. If the operand is a function pointer that points to a valid function, the result is a function designator. In either case, if the operand is of type *pointer to T*, the result is of type *T*. The operand must not be an incomplete pointer type, and the value must be a valid pointer.

Address Of Operator

The operator "address of" **&x** obtains the address of its operand, to produce an rvalue. If the operand is an lvalue, it must not have storage class **register** and it must not designate a **bitfield**. The result is a data object pointer. If the operand is a function designator, the result is a function pointer. In either case, if the operand is of type *T*, the result is of type *pointer to T*.

If ***x** is a valid expression, then **&*x** is always valid and compares equal to **x**. If **&x** is a valid expression, then ***&x** is always valid and designates the same data object or function as **x**.

Size Of Operator

The operator "size of" **sizeof x** produces an rvalue whose value is the number of bytes the compiler uses to represent the data type of its operand. The operand must be an lvalue, an rvalue, or a type name you write in parentheses, such as **sizeof (int [10])** or **sizeof (struct x)**. In any case, the compiler does not evaluate the operand, since it only needs to determine its type to determine the value of the result. It is always true that **sizeof (char)** has the value 1. On System/370, **sizeof 3** has the value 4 since the integer constant 3 is of type *int* and an *int* occupies four bytes.

The type of the operator **sizeof** is an unsigned integer type that can represent the size of the largest declarable data object. If you want your program to be portable, and you must declare a data object that can hold the result of **sizeof**, you should use the type definition **size_t**, described with the header file **<stdefs.h>** in Chapter 11, "C Library Reference." On System/370, the type of **sizeof** is *unsigned int*.

Type Cast Operators

A "type cast" operator is a type name you write in parentheses, such as **(char *)x** or **(float)x**. A type cast causes the compiler to convert the value of the operand to the type you name, as if it were assigning the value to a temporary data object of the named type. The operand must be an rvalue of scalar type. The result is an rvalue, whose value is the value that the compiler would have stored in the temporary data object. Its type is the named type, which the compiler widens to *int*.

The named type must be a scalar type. You may always write a type cast where the operand type is assignment compatible with the named type. Otherwise, one of the following must be true: If the named type is a function pointer, then the operand type must be a function pointer, not necessarily of the same type. If the named type is a data object pointer or incomplete pointer, then the operand type must be a data object pointer, an incomplete pointer, or an integer type. If the named type is an integer type, then the operand type must be a pointer type. The operations you can perform using pointer type casts are described earlier in this chapter in the section "Pointer Conversion." For example, **(double)3** has the value 3.0, and **(unsigned char)257** has the

value 1.

Multiplicative Operators

The multiplicative operators group left to right. The compiler may regroup expressions that use the operator "multiply," as described earlier in this chapter in the section "Regrouping."

Multiply Operator

The operator "multiply" **x*y** multiplies the value of **x** by the value of **y**, to produce an rvalue. The result type is the wider of the two operand types, which the compiler widens to *int*. Both operands must be rvalues of arithmetic type, which the compiler widens to the result type. For example, **3*4** has the value 12, and **3*4.0** has the value 12.0.

Divide Operator

The operator "divide" **x/y** divides the value of **x** by the value of **y**, to produce an rvalue. The result type is the wider of the two operand types, which the compiler widens to *int*. Both operands must be rvalues of arithmetic type, which the compiler widens to the result type. If the result is an integer type, the result value truncates toward zero. Do not divide by zero. For example, **7/4** has the value 1, and **7.0/4** has the value 1.75.

Remainder Operator

The operator "remainder" **x%y** obtains the remainder from dividing the value of **x** by the value of **y**, to produce an rvalue. The result type is the wider of the two operand types, which the compiler widens to *int*. Both operands must be rvalues of integer type, which the compiler widens to the result type. Do not divide by zero. If **x%y** is valid, it is always true that

```
(x/y)*y + (x%y)
```

is equal to **x**.

Additive Operators

The additive operators group left to right. The compiler may regroup expressions that use the operator "add," as described earlier in this chapter in the section "Regrouping."

Add Operator

The operator "add" **x+y** adds the value of **x** to the value of **y**, to produce an rvalue. Both operands must be rvalues. If both operands are of arithmetic type, the result type is the wider of the two operand types, which the compiler widens to *int*. In this case, the compiler widens both types to the result type. Otherwise, one operand must be a data object pointer and the other must be an integer type. In this case, the result type is the same as the data object pointer. The value is obtained as described earlier in this chapter in the section "Pointer Arithmetic." For example, 3+2 has the value 5, **3.0+2.7** has the value 5.7, and

"hello"+4 points at the last letter in the string.

Subtract Operator

The operator "subtract" **x-y** subtracts the value of **y** from the value of **x**, to produce an rvalue. Both operands must be rvalues. If both operands are of arithmetic type, the result type is the wider of the two operand types, which the compiler widens to *int*. In this case, the compiler widens both operands to the result type. If **x** is a data object pointer and **y** is an integer type, the result type is the same as the data object pointer. In this case, the value is obtained as described earlier in this chapter in the section "Pointer Arithmetic."

If neither of the above cases apply, both operands must be valid data object pointers of the same type that point to elements of the same array. In this case, the compiler obtains the value of the expression by dividing the difference between the two pointer values by the size in bytes of a data object of the type pointed to. The type of the result is a signed integer type having the same number of bits as the type of the operator **sizeof**. If you want your program to be portable, and you must declare a data object that can hold the difference between two pointers, you should use the type definition **ptrdiff_t**, described with the header file **<stdefs.h>** in Chapter 11, "C Library Reference." On System/370, this type is *int*. For example, **3-4** has the value **-1**, **2.7-1** has the value 1.7, and **a[7]-a[2]** has the value 5 for any **a** of type *array of T*.

Bitwise Shift Operators

The bitwise shift operators group left to right.

Left Shift Operator

The operator "left shift" **x<<y** shifts the value of **x** left the number of places you specify by the value of **y**, to produce an rvalue. Both operands must be rvalues of integer type. The compiler widens the **x** operand to *int*. The **y** operand must have a value greater than or equal to zero and less than the number of bits the compiler uses to represent the widened **x** operand. The result type is the type of the widened **x** operand. For example, **0x1234<<4** has the hexadecimal value 12340, and **0x1234<<0** has the hexadecimal value 1234.

Right Shift Operator

The operator "right shift" **x>>y** shifts the value of **x** right the number of place specified by the value of **y**, to produce an rvalue. Both operands must be rvalues of integer type. The compiler widens the type of **x** to *int*. The **y** operand must have a value greater than or equal to zero and less than the number of bits the compiler uses to represent the widened **x** operand. The result type is the type of the widened **x** operand. For example, **01234>>3** has the value 0123, and **01234>>0** has the value 01234. If you want your program to be portable, you should not right

shift a negative signed integer. On System/370, such a shift copies sign bits into vacant bit positions, but this behavior is not required of all implementations of C.

Relational and Equality Operators

The relational operators group left to right at the same level of precedence, and the equality operators group left to right at the next lower level of precedence. Do not let the compiler group these operators for you, however, because the result is often not useful. If you write `0<x<5`, for example, the expression looks like common notation for stating that `x` must lie between the values 0 and 5. However, the compiler groups this as `(0<x)<5`, which always has the value 1. This is because `(0<x)` becomes either 0 or 1, which is always less than 5. To test whether `x` lies between 0 and 5, you can use the operator logical AND described below to write `0<x&&x<5`.

Less Than Operator

The relational operator "less than" `x<y` compares the values of its operands, to produce an rvalue. Both operands must be rvalues. If the value of `x` compares less than the value of `y`, the value of the result is 1. Otherwise the value is 0. If both operands have arithmetic type, the compiler widens them to a common type. This common type is the wider of the two operand types, which the compiler widens to *int*. Otherwise, both operands must be valid data object pointers of the same type that point to elements of the same array. For example, `3<5` has the value 1, `3.0<3` has the value 0, and `&a[0]<&a[1]` has the value 1 for any `a` of type *array of T*. The type of the result is always *int*.

Less Than Or Equal To Operator

The relational operator "less than or equal to" `x<=y` compares the values of its operands to produce an rvalue. If the value of `x` compares less than or equal to the value of `y`, the value of the result is 1. Otherwise the value is 0. The rules for determining types are the same as for the operator "less than." For example, `5<=3` has the value 0, `3.0<=3` has the value 1, and `&a[0]<=&a[1]` has the value 1 for any `a` of type *array of T*.

Greater Than Operator

The relational operator "greater than" `x>y` compares the values of its operands to produce an rvalue. If the value of `x` compares greater than the value of `y`, the value of the result is 1. Otherwise the value is 0. The rules for determining types are the same as for the operator "less than." For example, `3>5` has the value 0, `3.0>3` has the value 0, and `&a[1]>&a[0]` has the value 1 for any `a` of type *array of T*.

Greater Than Or Equal To Operator

The relational operator "greater than or equal to" `x>=y` compares the values of its operands to produce an rvalue. If the value of `x`

compares greater than or equal to the value of **y**, the value of the result is 1. Otherwise the value is 0. The rules for determining types are the same as for the operator "less than." For example, **3>=5** has the value 0, **3.0>=3** has the value 1, and **&a[1]>=&a[0]** has the value 1 for any **a** of type *array of T*.

Equal To Operator

The equality operator "equal to" **x==y** compares the values of its operands to produce an rvalue. Both operands must be rvalues. If the value of **x** compares equal to the value of **y**, the value of the result is 1. Otherwise the value is 0. If both operands have arithmetic type, the compiler widens the operands to a common type. This common type is the wider of the two operand types, which the compiler widens to *int*. Both operands may be pointers of the same type, or one operand may be a *pointer to void* and the other a data object pointer. Otherwise, one operand must be a pointer type and the other must be an integer constant with the value 0. A pointer operand may be a valid pointer or a null pointer, which compares equal to integer 0. If two pointers point to the same function or data object, their values compare equal. You may compare any scalar type for equality or inequality with an integer constant with the value 0. For example, **3==5** has the value 0, **3.0==3** has the value 1, and **&a[0]==&a[1]** has the value 0 for any **a** of type *array of T*. The type of the result is always *int*.

Not Equal To Operator

The equality operator "not equal to" **x!=y** compares the values of its operands to produce an rvalue. If the value of **x** compares unequal to the value of **y**, the value of the result is 1. Otherwise the value is 0. The rules for determining types are the same as for the operator "equal to." For example, **3!=5** has the value 1, **3.0!=3** has the value 0, and **&a[0]!=&a[1]** has the value 1 for any **a** of type *array of T*.

Bitwise Binary Operators

The operator "bitwise AND" groups left to right. The operator "bitwise exclusive OR," at the next lower level of precedence, groups left to right. The operator "bitwise inclusive OR," at the next lower level of precedence, also groups left to right. The compiler may regroup expressions that use any of these operators, as described earlier in this chapter in the section "Regrouping".

Bitwise AND Operator

The operator "bitwise AND" **x&y** performs a logical AND between corresponding bits of its two operand values, to produce an rvalue. If both bits have the value 1, the result bit has the value 1. Otherwise the result bit has the value 0. The result type is the wider of the two operand types, which the compiler widens to *int*. Both operands must be rvalues of integer type, which the compiler widens to the result type. For example, **0x1100&0x1010** has the value 0x1000.

Bitwise Exclusive OR Operator

The operator "bitwise exclusive OR" **x^y** performs a logical exclusive OR between corresponding bits of its two operand values, to produce an rvalue. If the two bits differ, the result bit has the value 1. Otherwise the result bit has the value 0. The result type is the wider of the two operand types, which the compiler widens to *int*. Both operands must be rvalues of integer type, which the compiler widens to the result type. For example, **0x1100^0x1010** has the value 0x0110.

Bitwise Inclusive OR Operator

The operator "bitwise inclusive OR" **x|y** performs a logical inclusive OR between corresponding bits of its two operand values, to produce an rvalue. If either bit has the value 1, the result bit has the value 1. Otherwise the result bit has the value 0. The result type is the wider of the two operand types, which the compiler widens to *int*. Both operands must be rvalues of integer type, which the compiler widens to the result type. For example, **0x1100|0x1010** has the value 0x1110.

Logical and Conditional Operators

The operator "logical AND" groups left to right. The operator "logical OR," at the next lower level of precedence, groups left to right. The operator "conditional," at the next lower level of precedence, groups right to left. Each of these operators imposes some restriction on the order in which the compiler evaluates its operands during program execution. You can use them to write expressions that have predictable side effects, or that avoid evaluating operands that are invalid.

Logical AND Operator

The operator "logical AND" **x&&y** compares the values of its operands against 0, to produce an rvalue. If the **x** operand is 0, the result has the value 0 and your program does not evaluate the **y** operand. Otherwise, the result has the value 1 if the **y** operand is nonzero, or the value 0 if the **y** operand is 0. Each operand can be any scalar type, which may differ from the type of the other operand. For instance, **0&&printf("!")** has the value 0 (**printf** is not called), **1&&0** has the value 0, and **8&&3.2** has the value 1. The result type is always *int*.

Logical OR Operator

The operator "logical OR" **x||y** compares the values of its operands against zero, to produce an rvalue. If the **x** operand is nonzero, the result has the value 1 and your program does not evaluate the **y** operand. Otherwise, the result has the value 1 if the **y** operand is nonzero, or the value 0 if the **y** operand is zero. Each operand can be any scalar type, which may differ from the type of the other operand. For instance, **1||printf("!")** has the value 1 (**printf** is not called), **3<2||3<1** has the value 0, and

`8||3.2` has the value 1. The result type is always *int*.

Conditional Operator

The operator "conditional" `x?y:z` compares its `x` operand against zero, to determine which of the operands `y` and `z` to evaluate. If the `x` operand is nonzero, your program evaluates `y` and not `z`. Otherwise, your program evaluates `z` and not `y`. The `x` operand can be any scalar type. If `y` and `z` are both void expressions, the result is a void expression. Otherwise, both operands must be rvalues and the result is an rvalue. If both `y` and `z` are of arithmetic type, the result type is the wider of the two types, which the compiler widens to *int*. The compiler widens both operands to the result type. If both `y` and `z` have the same type, the result is the common type. You may use structure types as well as scalar types. If one of the two operands is *pointer to void*, the other may be a data object pointer. In this case, the data object pointer is converted to *pointer to void*, which is the type of the result. Otherwise, one of the two operands must be a pointer type and the other an integer constant expression with the value 0. In this case, the compiler converts the integer 0 to the pointer type, which is the type of the result. For example, `1?5:8` has the value 5, `3<4?2.7:5` has the value 2.7, and `0?0:"abc"` points at the string.

Assignment Operators

The assignment operators all have an important side effect. They replace the value stored in a data object with another value. Each also produces an rvalue, so you can use it as the operand of another operator, as in `x+(y=z)`. In all cases, the value of the result is the new value stored in the data object. The type of the result is the type of the data object, which the compiler widens to *int* if the type is arithmetic. Your program will store the new value in the data object before using the result of the operator. The assignment operators group right to left.

In all the examples below, `x` is of type *int* and has the initial stored value 3.

Gets Operator

The assignment operator "gets" `x=y` stores the value of `y` in the data object designated by `x`. The `x` operand must be a modifiable lvalue. Its previous stored value need not be valid. The `y` operand must be an rvalue that is assignment compatible with the type of `x`. You can assign structure types as well as scalar types. If the types of `x` and `y` differ, then the compiler converts the value of `y` as described earlier in this chapter in the section "Arithmetic Conversions".

Gets Multiplied Operator

The assignment operator "gets multiplied" `x*=y` stores the current value of `x`, multiplied by the value of `y`, in the data object designated by `x`. Operand type restrictions and conversions are the

same as for **x*y**. The compiler converts the result of the multiplication to the type of **x** before storing it. For example, **x*=2** has the value 6, and **x*=5.1** has the value 15.

Gets Divided Operator

The assignment operator "gets divided" **x/=y** operates in the same way as **x*=y**, except that it computes the value **x/y**. For example, **x/=2** has the value 1, and **x/=-0.5** has the value -6.

Gets Remainder Operator

The assignment operator "gets remainder" **x%=y** operates in the same way as **x*=y**, except that it computes the value **x%y**. For example, **x%=2** has the value 1, and **x%=-2** has the value 1.

Gets Added Operator

The assignment operator "gets added" **x+=y** operates in the same way as **x*=y** except that it computes the value **x+y**. The **y** operand cannot be a data object pointer. For example, **x+=3** has the value 5, and **x+=5.7** has the value 8.

Gets Subtracted Operator

The assignment operator "gets subtracted" **x-=y** operates in the same way as **x*=y**, except that it computes the value **x-y**. The **y** operand cannot be a data object pointer. For example, **x-=3** has the value 0, and **x-=5.7** has the value -2.

Gets Left Shifted Operator

The assignment operator "gets left shifted" **x<<=y** operates in the same way as **x*=y**, except that it computes the value **x<<y**. For example, **x<<=3** has the value 24, and **x<<=0** has the value 3.

Gets Right Shifted Operator

The assignment operator "gets right shifted" **x>>=y** operates in the same way as **x*=y**, except that it computes the value **x>>y**. For example, **x>>=1** has the value 1, and **x>>=0** has the value 3.

Gets AND Operator

The assignment operator "gets AND" **x&=y** operates in the same way as **x*=y**, except that it computes the value **x&y**. For example, **x&=3** has the value 3, and **x&=4** has the value 0.

Gets Exclusive OR Operator

The assignment operator "gets exclusive OR" **x^=y** operates in the same way as **x*=y**, except that it computes the value **x^y**. For example, **x^=3** has the value 0, and **x^=4** has the value 7.

Gets Inclusive OR Operator

The assignment operator "gets inclusive OR" **x|=y** operates in the same way as **x*=y**, except that it computes the value **x|y**. For

example, `x|=3` has the value 3, and `x|=4` has the value 7.

Comma Operator

The comma operator has the lowest level of precedence. You use it to write a series of expressions in a context where only one expression is permitted, as in

```
if (must_swap)
    temp = x, x = y, y = temp;
```

An *if* statement controls the execution of one statement, which may be an *expression* statement. Here, the comma operator is used twice to group into one expression three closely related expressions. The comma operator also evaluates its operands left to right in all cases, so you can depend on the assignments happening in the order you expect. The comma operator groups left to right.

The operator "comma" `x,y` evaluates its operands in a specified order. First the `x` operand is evaluated as a void expression. Then the `y` operand is evaluated. If `y` is a void expression, the result is a void expression. Otherwise, the result is an rvalue whose value is the value of `y`. If `y` is an arithmetic type, the compiler widens it to *int*. The `y` operand may be a structure type as well as a scalar. The type of the result is the widened type of `y`.

You also use commas to separate the arguments in a function call. If you write `f(g(),x)`, for example, the compiler always assumes you are writing two arguments, not one argument which is the result of applying the comma operator. If the comma operator is what you intend, however, you must put extra parentheses around the expression, as in `f((g(),x))`.

Constant Expressions

Sometimes the compiler restricts the expressions you can write. If you are declaring the size of an array, for example, the compiler must determine the value of an integer expression. In the case of the expression

```
char a[5*4+3];
```

the compiler recognizes that you want the array `a` to have 23 elements. The compiler performs as much arithmetic as it can while it is translating your source file. This permits you to write complex expressions in places where the compiler must know the value at compile time.

A "constant expression" is one that the compiler can reduce to a known value before it reads any more of your source file. The most restrictive form is a "constant integer expression," which the compiler requires in a variety of contexts. If you write an expression of integer type, and use only certain operands and operators, you can be sure that it is a constant integer expression. If it is not, and the compiler requires one, the compiler will emit

an error message.

You can have any expression as the operand of the operator **sizeof**. For all other operands, you can use: integer constants, floating point constants, character constants, and enumeration constants. You can write type casts that convert to arithmetic type. You can also use the operators: logical NOT **!x**, bitwise NOT **~x**, plus **+x**, minus **-x**, size of **sizeof x**, multiply **x*y**, divide **x/y**, remainder **x%y**, add **x+y**, subtract **x-y**, left shift **x<<y**, right shift **x>>y**, less than **x<y**, less than or equal **x<=y**, greater than **x>y**, greater than or equal **x>=y**, equals **x==y**, not equal **x!=y**, bitwise AND **x&y**, bitwise exclusive OR **x^y**, bitwise inclusive OR **x|y**, logical AND **x&&y**, logical OR **x||y**, and conditional **x?y:z**. The following are examples of constant integer expressions, assuming that **red** and **yellow** are enumeration constants:

```
3
5 + 6
(int)(2.7 * 86.4 + 'a')
red < yellow ? red + 1 : red
```

If you are writing a static initializer for an arithmetic data object, your program is subject to fewer restrictions. The constant expression can have any arithmetic type. The compiler converts its value to the type of the data object, using the rules described earlier in this chapter in the section "Arithmetic Conversions".

If you are writing a static data initializer for a data object of some pointer type, your program is subject to even fewer restrictions. A "constant pointer expression" can be an integer constant expression with the value zero, or it can have the same pointer type as the data object. You can use identifiers that name functions and data objects with static lifetimes as operands. You can write type casts of pointer type. You can use the operators: subscript **x[y]**, points at **x->y**, select member **x.y**, indirect on ***x**, and address of **&x**. None of these must access a stored value, however. The final value must be expressible as an integer constant, as the address of a function or data object, or as the address of a data object plus or minus an integer constant. The following examples of constant pointer expressions could be used to initialize a data object of type *pointer to char*, assuming that **a** is an array of any type *T* with static lifetime:

```
0
"abc"
(char *)a
(char *)&a[4]
```

The preprocessor directive **#if** evaluates constant integer expressions, with a number of additional restrictions. These are discussed in Chapter 7, "The Preprocessor."

Chapter 6: Statements

The statements in a C function perform actions and determine the flow of control through a function. When you call a function, control passes to the first statement of that function. The first statement performs its defined action, then passes control to a successive statement. The successive statement is normally the next statement in sequence, but some statements alter the normal flow of control. Control passes from statement to statement until a *return* statement gets control. The function then returns control to the expression that called it.

You write a statement with various keywords, punctuation, expressions, and other statements. Except for a *compound* statement, every statement either ends with a semicolon or ends with a statement that does so. The net effect is that you end each statement you write with a semicolon. You can treat any sequence of statements as a single statement by writing braces around the sequence.

Test and Void Expressions

Many statements conditionally alter flow of control by evaluating a "test expression." A test expression is an rvalue of scalar type. If the value of the expression does not compare equal to zero, the test is "true." Otherwise, the value of the expression compares equal to zero, and the test is "false." You can compare *any* scalar expression against zero. Some examples of test expressions are:

```
x < 0
'0' <= c && c <= '9'
p          /* same as p != 0 */
1          /* always true */
```

Another form of expression that occurs often in statements is a void expression. This may be an expression of type *void*, which produces no value. It may also be an rvalue of any type whose value is discarded. You evaluate a void expression for its side effects, such as calling a function or altering the value stored in a data object. Some examples of void expressions are:

```
srand(0)        /* srand returns void */
printf("hello\n")  /* printf returns int, discarded */
x = 3           /* assign new value to x */
++x             /* increment stored value */
y = 2, ++x      /* comma lets you do two things */
```

Chapter 5, "Expressions," describes both scalar rvalues and void expressions.

Labels

You write a label at the beginning of a statement so that your program can transfer control to the statement by means of a *goto* statement or a *switch* statement. There are three kinds of labels: "plain labels," "*case* labels," and "*default* labels."

Plain Label

A "plain label" consists of an identifier followed by a colon. You implicitly declare an identifier as a label when you write a plain label, or when you write a *goto* statement that contains the identifier. Chapter 3, "Identifiers," discusses the scope, visibility, and name space of labels. When your program transfers control to a *goto* statement, it transfers control unconditionally to the statement within the same function that has a plain label with a matching identifier. Some examples of plain labels are:

```
    i = 0;
top:
    if (i < sizeof (a) / sizeof (a[0])
        goto bottom;
    printf("a[%i] = %i\n", i, a[i]);
    goto top;
bottom:  ;
```

Case Label

A "*case* label" consists of the keyword **case**, followed by a constant integer expression, followed by a colon. You write a *case* label only within a *switch* statement. When your program evaluates the expression in the *switch* statement, it transfers control to the statement following a *case* label if the value of the expression in the *case* label compares equal to the value of the expression in the *switch* statement.

Default Label

A "*default* label" consists of the keyword **default** followed by a colon. You write a *default* label only within a *switch* statement. When your program evaluates the expression in the *switch* statement, it transfers control to the statement following a *default* label if the value of the expression in the *switch* statement compares equal to no *case* label value.

The section "switch Statement," later in this chapter, discusses *case* labels and *default* labels in more detail. An example is:

```
switch (p->state)
     {
case READING:
     /* enum {READING, WRITING, CLOSED, .....} */
case WRITING:
     printf("can't reopen %s\n", p->name);
     break;
case CLOSED:
     p->state = READING;
     break;
default:
     printf("inconsistent state!\n");
     p->state = CLOSED;
     }
```

If control passes to a labelled statement from the preceding statement in sequence, the label causes no action and has no effect on flow of control. You can write any number of labels, of various kinds, before a statement.

Kinds of Statements

This chapter describes all the statements of C, in the order listed below:

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

expression Statement

expr;

An *expression* statement consists of an expression, followed by a semicolon. It evaluates the void expression *expr* and passes control to the next statement in sequence. You use *expression* statements to call functions and to store values in data objects.

Some examples are:

```
root1 = disc - b / (2.0 * a);
printf("are you sure? ");
++x;
x = cos(theta), y = -sin(theta);
```

null Statement

```
;
```

A *null* statement consists of a semicolon standing alone. It does nothing but pass control to the next statement in sequence. You use a *null* statement where you must write a statement, but want to perform no additional action.

Some examples are:

```
while (getchar() != '\n')    /* consume rest of line */
    ;
    top: ;          /* null statement holds a label */
```

compound Statement

```
{
declarations

statements
}
```

A *compound* statement consists of a sequence of declarations followed by a sequence of statements, all enclosed in braces. It creates a block with a new scope, allocates data objects with dynamic lifetime, initializes any such data objects, and then passes control to the first of the sequence of statements. If the last of the sequence of statements passes control to the next statement in its sequence, then the *compound* statement passes control to the next statement in its sequence. You use a *compound* statement to introduce a new scope block, or to group a sequence of statements in a context that would otherwise permit only one.

You may write a *goto* statement or a *switch* statement that transfers control to a statement within the *compound* statement. In this case, the *compound* statement still creates a new scope block and allocates data objects with dynamic lifetime, but it does not initialize any such data objects. On System/370, your program allocates all data objects with dynamic lifetime upon function entry. Your program executes *no* code to allocate data objects upon entry to each block.

Some examples of *compound* statements are:

```
if (d < 0)
    {
    d = -d;
    minus = YES;
    }
```

```
if (compare(px, py) < 0)
     {
     register int i;
     Item t;

     for (i = 0; i < sizeof (*px); ++i)
          t = px[i], px[i] = py[i], py[i] = t;
     }
```

return Statement

return *expr*;

A *return* statement consists of the keyword **return**, followed by an expression, followed by a semicolon. If the expression *expr* is present, your program evaluates it and converts its value to the type returned by the function. The expression must be an rvalue that is assignment compatible with the type returned by the function. In all cases the *return* statement terminates execution of the function, restores the previous calling environment, and returns control to the expression that called it. You use a *return* statement to return control from a function and to specify a return value.

If the expression *expr* is present, the converted value becomes the value of the function call in the calling expression. If the expression *expr* is not present, the calling expression must be a void expression. If the function is a void function, the calling expression must be a void expression and the expression *expr* must not be present.

At the end of every function body is an implicit:

```
return;
```

Some examples of *return* statements are:

```
return;
return cos(x);
return (a < 0 ? x + 1 : x - 1);
```

if Statement

if (*test*)
     statement

An *if* statement consists of the keyword **if**, followed by a test expression in parentheses, followed by a controlled statement. If *test* is true, control passes to the controlled statement and then to the next statement in sequence. Otherwise, control passes directly to the next statement in sequence. You use an *if* statement to execute a controlled statement at most once, under control of a test.

Some examples of *if* statements are:

```
if (x < 0)
     x = -x;
if (abs(val) < EPSILON)
     return (answer);
```

if/else Statement

```
if (test)
     statement
else
     statement
```

An *if/else* statement consists of an *if* statement followed by the keyword **else** and a second controlled statement. If *test* is true, control passes to the controlled statement following the *test* and then to the next statement in sequence. Otherwise, control passes to the second controlled statement following the **else** and then to the next statement in sequence. You use an *if/else* statement to execute exactly one of two alternate statements, under control of a test.

The first controlled statement must not be an *if* statement, and it must not end in a controlled *if* statement. Otherwise, the compiler will group the **else** with the controlled statement, as in:

```
if (i < 0)
     if (j < 0)
          printf("both negative\n");
else           /* INDENTING IS MISLEADING */
     printf("i is not negative\n");    /* NOT TRUE! */
```

You can write this example correctly by enclosing the controlled *if* statement in braces, as in:

```
if (i < 0)
     {
     if (j < 0)
          printf("both negative\n");
     }
else
     printf("i is not negative\n");    /* now true */
```

Some examples of *if/else* statements are:

```
if (x < y)
     small = x, large = y;
else
     small = y, large = x;
if (reading)
     n = read_file(buf, size);
else
     n = write_file(buf, size);
```

else/if Chain

```
if (test)
     statement
else if (test)
     statement
else
     statement
```

An *else/if* chain consists of an *if/else* statement whose second controlled statement is another *if/else*. If the first *test* is true, control passes to the first controlled statement and then to the next statement in sequence. Otherwise, if the second *test* is true, control passes to the second controlled statement and then to the next statement in sequence. Otherwise, control passes to the third controlled statement and then to the next statement in sequence. You can extend the chain to arbitrary depth. You use an *else/if* chain to execute exactly one of a series of statements, under control of a series of tests.

None of the controlled statements, except the last, can be an *if* statement or may end in a controlled *if* statement.

Some examples of *else/if* chains are:

```
if (x < 0)
     return (-fun(-x));
else if (x == 0)
     return (0);
else
     return (approx(x));

if (isdigit(ch))
     do_num(ch);
else if (isletter(ch))
     do_name(ch);
else if (ispunct(ch))
     do_ops(ch);
else
     printf("unknown character\n");
```

while Statement

```
while (test)
     statement
```

A *while* statement consists of the keyword **while**, followed by a test expression in parentheses, followed by a controlled statement. If *test* is true, control passes to the controlled statement and then back to the evaluation of *test*. Otherwise, control passes directly to the next statement in sequence. You use a *while* statement to execute a controlled statement zero or more times, under control of a test.

Some examples of *while* statements are:

```
while ((c = getchar()) != EOF)
    putchar(c);

while (EPS < abs(d - ans * ans))
    ans = (d / ans + ans) / 2;
```

do/while Statement

```
do
    statement
    while (test);
```

A *do/while* statement consists of the keyword **do**, followed by a controlled statement, followed by the keyword **while**, followed by a test in parentheses, followed by a semicolon. Control first passes to the controlled statement and then to the evaluation of *test*. If *test* is true, control passes back to the controlled statement. Otherwise control passes directly to the next statement in sequence. You use a *do/while* statement to execute a controlled statement one or more times, under control of a test.

Some examples of *do/while* statements are:

```
do
    putchar(' '); /* put one or more spaces */
    while (++col < tabstop);

do  {
    est = (x / est + est) / 2.0;
    err = dabs(est) / x;
    } while (EPS < err);
```

for Statement

```
for (init; test; reinit)
    statement
```

A *for* statement consists of the keyword **for**, followed by three expressions within the same set of parentheses, followed by a controlled statement. You separate the three expressions by semicolons. The first expression *init* is a void expression, the second expression *test* is a test expression, and the third expression *reinit* is a void expression. The first expression *init* is evaluated once when the *for* statement first gets control. Then if the second expression *test* is true, control passes to the controlled statement, to the evaluation of the third expression *reinit*, and then back to the evaluation of the second expression *test*. Otherwise, control passes directly to the next statement in sequence. You use a *for* statement to execute a controlled statement zero or more times, under control of a test, and to express all loop control expressions in one statement.

Some examples of *for* statements are:

```
for (i = 0; i < sizeof (a); ++i)  /* walk an array */
    a[i] = b[i + 1];
```

```
for (p = first; p; p = p->next)   /* walk a list */
     if (p->value == value)
          break;
```

switch Statement

```
switch (expr)
     {
     declarations

case case_expr:
     statements
     .....
default:
     statements
     }
```

A *switch* statement consists of the keyword **switch**, followed by an expression in parentheses, followed by a controlled *compound* statement. Your program evaluates the integer rvalue *expr* once, and then compares its value against the value of *case_expr* for each of the *case* labels within the *switch* statement. If the values compare equal for one of the *case* labels, then control transfers to the statement immediately following that *case* label. If none of the values compare equal and there is a *default* label, then control transfers to the statement immediately following the *default* label. Otherwise control transfers to the statement following the *switch* statement. You use a *switch* statement to transfer control to one of several labelled statements within a controlled statement, under control of an integer expression.

Each of the expressions *case_expr*, within each of the *case* labels, must be a constant integer expression. The compiler evaluates each of the expressions *case_expr* and converts it to the type of the expression *expr*. All expressions *case_expr* within a given *switch* statement must have distinct values after conversion. You can write at most one *default* label in a *switch* statement. You can prefix any statement within the controlled *compound* statement of a *switch* statement with a *case* label or a *default* label. These labels may be inside any contained statement, within the controlled *compound* statement, except a contained *switch* statement.

If you write any declarations at the beginning of the *compound* statement, the *switch* statement will allocate data objects you declare there but it will not initialize any such data objects with dynamic lifetime. Once control transfers to a statement within the *compound* statement, the normal flow of control occurs. Write *break* statements to terminate each sequence of statements you write after a *case* label or a *default* label, except the last sequence. Some examples of *case* statements are:

```
switch (color)
     {
case RED:
case ORANGE:
case YELLOW:
     type = WARM;
     break;
case GREEN:
     type = COOL;
     }

switch (c = getchar())
     {
case EOF:
     return;
case 'l':
     go_left();
     break;
case 'r':
     go_right();
     break;
default:
     put_it(c);
     }
```

break Statement

```
break;
```

A *break* statement consists of the keyword **break** followed by a semicolon. It passes control to the next statement after the innermost nested *switch*, *while*, *do/while*, or *for* statement. You use a *break* statement to terminate a *switch* or a loop early.

Some examples of *break* statements are:

```
for (p = root; p != NULL; p = p->next)
     if (p->value == value)
          break;

switch (code)
     {
case 0x17ff3b2:
     mode = OLD_STYLE;
     break;
case 0x3322232:
     mode = NEW_STYLE;
     break;
default:
     mode = VARYING;
     }
```

continue Statement

continue;

A *continue* statement consists of the keyword **continue** followed by a semicolon. It passes control to the statement that follows the controlled statement in the innermost *while*, *do/while*, or *for* statement. That successive statement is the test expression in a *while* or a *do/while* statement, or the third expression in a *for* statement. You use a *continue* statement to repeat the test expression in a loop early.

An example of the *continue* statement is:

```
for (p = root; p != NULL; p = p->next)
     {
     if (p->value != value)
          continue;
     process(p);
     }
```

goto Statement

goto *label*;

A *goto* statement consists of the keyword **goto**, followed by a label, followed by a semicolon. It passes control to the statement prefixed by the plain label whose identifier matches *label*. You use a *goto* statement to bypass the flow of control that other statements provide.

An example of the *goto* statement is:

```
     for (p = root; p != NULL; p = p->next)
          if (p->value == value)
               for (q = p->list; q; q = q->next)
                    if (q->flavor == flavor)
                         goto done;
done: ;   /* here on double match (p != NULL) or no match */
```

Chapter 7: The Preprocessor

Part of the compiler is a preprocessor which includes the contents of specific source files you name, conditionally skips portions of your source text, and lets you define source text "macros." It performs these functions after the compiler groups input into tokens and before it groups tokens into declarations. Chapter 2, "Elements of the C Language," describes how you write tokens. Chapter 4, "Declarations," describes how you write declarations.

The preprocessor evaluates your C source file one text line at a time. If the first character other than whitespace on a text line is the character #, the text line is a "preprocessor directive." The next token on the text line must be an identifier, which is the name of the preprocessor directive. Write only the preprocessor directive names described in this chapter. You cannot introduce new preprocessor directive names. Preprocessor directive names are *not* keywords or reserved identifiers. A preprocessor directive may appear anywhere within your compilation.

The preprocessor either skips or "expands" any text line other than a preprocessor directive. It skips text lines under control of "conditional preprocessor directives." It expands text lines by replacing each identifier that is defined as a macro with the defining text for the macro.

This chapter describes all of the preprocessor directives. It covers the following topics:

file inclusion preprocessor directives: How you include source files

conditional preprocessor directives: How you conditionally skip portions of your source text

macro preprocessor directives: How you define macros, and how the preprocessor expands them

information preprocessor directives: How you communicate special information to the compiler.

File Inclusion Preprocessor Directives

The preprocessor directives

#include <*fname*>

and

#include "*fname*"

tell the preprocessor to include the contents of a file whose name is determined from *fname*, in place of the preprocessor directive itself. Enclose *fname* either in angle brackets **< >** or in double quotation marks **" "**, as shown. Files you include this way may themselves contain **#include** preprocessor directives. You may nest **#include** preprocessor directives to a depth of at least four files.

No source file you include may end with an incomplete text line, within a comment, within a conditional preprocessor directive group, or within a macro expansion. Conditional preprocessor directive groups and macro expansions are described later in this chapter.

The preprocessor uses *fname* to construct one or more file names. How it constructs file names depends upon which form of the **#include** preprocessor directive you write, and which operating system you are using. The compiler may use a "search modifier" to construct a file name. You specify the search modifier when you invoke the C compiler. If the preprocessor cannot open an include file for reading, the compiler emits an error message. The *C Compiler User's Guide* for your system describes how to form correct file names, tells you the default value for the search modifier for each form of the **#include** preprocessor directive, and tells you how you can change the search modifier when you invoke the compiler.

Rules for #include File Processing

You enclose *fname* in double quotation marks if you want the preprocessor to include one of your own files rather than a file shared by more than one user. An example is:

```
#include "myhdr.h"
```

If you enclose *fname* in angle brackets, the preprocessor will include the file from the include library provided with the C compiler. This form of **#include** preprocessor directive is usually used to include project wide include files or one from the include library provided by the C compiler. An example is:

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

On VM/CMS, the compiler command options **SEarch(*mod*)** and **LSEarch(*mod*)** allow you to specify a search modifier *mod* for preprocessor **#include** files you enclose in angle brackets or double quotation marks within your program. *mod* represents the minidisk at which the compiler will begin the search for your include file. For example, if you specify the compiler command option **SE(x)**, the compiler will search minidisks **X**, **Y**, and **Z** for include files you enclose in angle brackets. It will not search minidisks **A** through **W**. The table below explains how the compiler performs **#include** file processing on VM/CMS depending on how you write your include file names.

<test.h>	The compiler will search as directed by the **SEarch(*mod*)** option if you specify it. Otherwise it will search all accessed minidisks (normal VM/CMS

search order).

<test.h.b> The compiler will get the file **TEST H B**.

<test.h.*> The compiler will search all accessed minidisks (normal VM/CMS search order).

"test.h" The compiler will search all minidisks as indicated by the **LSEarch(***mod***)** option if you specify it. Otherwise it will search all accessed minidisks (normal VM/CMS search order). If the compiler does not find the file using the above rules, it will search for the file as if you wrote its name as **<test.h>**.

"test.h.b" The compiler will get the file **TEST H B**.

"test.h.*" The compiler will search all accessed minidisks (normal VM/CMS search order).

On MVS and MVS/XA, the compiler command options **SEarch(***name***)** and **LSEarch(***name***)** allow you to specify a name *name* which specifies one of the following:

1. The name of a partitioned dataset or
2. A DDname of the form **dd:***name*, where *name* is a DDname defining a partitioned dataset.

For example, **'jones.local'**, **local**, and **dd:local** are valid definitions for *name* provided that **local** is a DDname assigned to a partitioned dataset. For information about file names on MVS and MVS/XA, see the description of the **fopen** function in Chapter 11, "C Library Reference."

The LIBH Option and Partitioned Dataset Member Names under MVS

By default under MVS and MVS/XA, include file names that have the suffix '**.h**' refer to member names that have the suffix '**$H**'. The **LIBH** compiler command option directs the compiler to strip the '**.h**' suffix from all include file names that refer to partitioned dataset members. Stripping off the '**.h**' suffix allows you to specify up to two more significant characters in include file names. When you specify **LIBH** your include file names may be up to eight characters in length instead of the default maximum length of six characters. When you specify **LIBH**, an include file name you write as **<test.h>** can refer to the member name **TEST**. The table below explains how the compiler performs **#include** file processing on MVS and MVS/XA, depending on how you write the include file names. These examples assume that **NOLIBH** has been specified.

<test.h> The compiler will get the member **TEST$H** from the dataset you reference with the **SEarch(***name***)** compiler command option, if you specify it. If the member **TEST$H** is not in the dataset you reference with **SEarch(***name***)** option, or if you do not specify **SEarch(***name***)**, the compiler will get the member **TEST$H** from the dataset referenced by the DDname

INCLUDE.

"test.h" The compiler will get the member **TEST$H** from the dataset you reference with the **LSEarch(***name***)** compiler command option, if you specify it. If you do not specify **LSEarch(***name***)**, the compiler will get the sequential dataset **test.h**. If the compiler does not find the file using these rules, it will search for the file as if you wrote its name as **<test.h>**.

File Name Macro Expansion

You can also write an **#include** preprocessor directive as

#include *ident*

The identifier *ident* must be a macro that expands to one of the two forms shown earlier.

Do not confuse file names enclosed in double quotation marks with string constants. Escape sequences are not processed and string concatenation does not occur before the file name is constructed. It is the original "spelling" of the **#include** preprocessor directive, or of the macro expansion of the identifier *ident*, that determines the file name.

Conditional Preprocessor Directives

Several preprocessor directives allow you to specify conditions under which the preprocessor will skip parts of your source file or an include file.

Testing for Macro Definition

The preprocessor directive

#ifdef *ident*

tells the preprocessor not to skip the text lines immediately following this directive if the identifier *ident* is currently defined as a macro. Otherwise, the preprocessor skips text lines and preprocessor directives until it encounters the preprocessor directive

#endif

later in the same file. The range of lines from the **#ifdef** preprocessor directive to its corresponding **#endif** preprocessor directive is a "conditional preprocessor directive group." If the preprocessor encounters any conditional preprocessor directive group while skipping, the entire group is skipped.

For example:

```
#ifdef    FLOAT_FORM     /* skip if FLOAT_FORM not defined */
typedef double LARGE;

#ifdef    DO_DIVIDES     /* skip if either not defined */
extern LARGE divide(LARGE, LARGE);
#endif                   /* for DO_DIVIDES */

#endif                   /* for FLOAT_FORM */
```

You can use **#ifdef** to skip a **#include** preprocessor directive that would fail if not skipped. All preprocessor directives that the preprocessor skips must be formatted properly, however.

The preprocessor directive

#ifndef *ident*

behaves exactly like **#ifdef**, except that the preprocessor skips the text lines that follow if the identifier **ident** is defined as a macro.

Testing for Arithmetic Value

The preprocessor directive

#if *expr*

behaves like **#ifdef**, except that the preprocessor skips text lines if the expression *expr* evaluates to zero. The expression must be a constant integer expression, with the following added properties:

* The preprocessor cannot evaluate the operator **sizeof** or any type cast operators.
* The preprocessor cannot evaluate any floating constants or enumeration constants.
* The preprocessor converts all integer constants, and all intermediate results, to type *long*.
* The preprocessor replaces either of the sequences

  defined *ident*

  or

  defined (*ident***)**

  with the value 1 if *ident* is currently defined as a macro, and with 0 otherwise.
* The preprocessor expands any other identifier defined as a macro, following the usual rules described later in this chapter.
* The preprocessor replaces any other identifier *not* defined as a macro with the value 0.

For example:

```
#define MACH_TYPE  370
#if MACH_TYPE == 370 || defined DO_DIVIDES
.... /* don't skip if 370 or doing divides */
```

The form

#if defined *ident*

is equivalent to

#ifdef *ident*

and the form

#if !defined *ident*

is equivalent to

#ifndef *ident*

The C language retains **#ifdef** and **#ifndef** preprocessor directives primarily for compatibility with older programs. Use the **#if** directive exclusively.

Alternate Groups

The preprocessor directive

#else

lets you specify an alternate group of text lines within a conditional preprocessor directive group. If skipping is in effect when it encounters **#else**, it stops. Otherwise, skipping begins until the corresponding **#endif**. Write an **#else** preprocessor directive only inside a conditional preprocessor directive group. For example:

```
#if MACH_TYPE == 370
.... /* don't skip if 370 */
#else
.... /* skip if 370 */
#endif
```

The preprocessor directive

#elif *expr*

behaves much as

#else
#if *expr*

except that you write only one **#endif** preprocessor directive to end the conditional preprocessor directive group. For example:

```
#if MACH_TYPE == 370
.... /* don't skip if 370 */
#elif MACH_TYPE == 36
.... /* don't skip if 36 */
#else
.... /* skip if either 370 or 36 */
#endif
```

You may write as many **#elif** preprocessor directives as you wish within one conditional preprocessor directive group. You can write at most one **#else** preprocessor directive after the last of any **#elif** preprocessor directives. Your choices are very similar to

those you have when writing executable statements within a function body. The corresponding statements *if*, *if/else*, and *else/if* are described in Chapter 6, "Statements."

Macro Preprocessor Directives

The preprocessor directive

#define *ident def*

defines the identifier *ident* as a macro. In text lines that follow, the preprocessor will replace the identifier *ident* with the expansion *def*. Do not define an identifier that is currently defined as a macro, unless you define it with a definition *def* spelled exactly the same as its current definition.

If the expansion *def* contains any instance of *ident*, the preprocessor leaves it alone. The preprocessor will, however, expand as usual any other identifiers defined as macros, except that any instances of *ident* within their expansions is left alone. For example:

```
#define PI        pi_object
#define pi_object +(pi_object)
x = PI;            /* expands to: x = +(pi_object) */
```

This sequence defines the macro **PI** as the data object **pi_object**. The second macro ensures that only the value of the data object is available from that point on in the compilation.

Macros with Arguments

The preprocessor directive

#define *ident*(*args*) *def*

defines the identifier *ident* as a macro with arguments. In text lines that follow, the preprocessor will replace the identifier *ident* followed by an argument list in parentheses with the expansion *def*. Do not define an identifier that is currently defined as a macro, unless you define it with a definition *def* spelled exactly the same as its current definition.

Write *no* whitespace between *ident* and the left parenthesis. This is the *only* context in the C language where the whitespace between distinguishable tokens is significant. *args* consists of zero or more "argument identifiers" separated by commas. All argument identifiers must be distinct. Their scope is the expansion *def*. For example:

```
#define sum(x, y)  (x + y)   /* has two arguments */
#define product    (x * y) /* has NO arguments */
```

The preprocessor will expand an identifier defined as a macro with arguments only if you write an actual argument list in parentheses immediately following the identifier in your source text. You may write whitespace between the macro identifier and the left parenthesis that follows. You may write *any* sequence of tokens

for each actual argument, provided that parentheses balance in the sequence and any commas are inside parentheses. The actual arguments are separated by commas.

Write the same number of actual arguments as you write argument identifiers in the **#define** preprocessor directive. You may write the macro identifier and actual arguments across multiple text lines, with the following constraints:

* Write *no* preprocessor directives among the text lines.
* All of the text lines must be within the same file.
* All of the text lines must total no more than 511 characters, counting one character for a newline at the end of each line.

After identifying the actual arguments, the preprocessor expands each of them. The macro that is being expanded is still defined and may be expanded when the arguments are expanded. For example:

```
#define sum(x, y)  (x + y)
     sum(sum(3, 2), 4)  /* same as ((3 + 2) + 4) */
```

The expansion *def* replaces the macro identifier and actual arguments by the following rules:

* The operator # followed by an argument identifier is replaced by a "constructed string constant," as described below.
* An argument identifier is replaced by its expansion.
* The macro identifier *ident* is left alone.
* Any other macro identifiers are expanded as usual, except that the macro identifier is left alone.
* Any other tokens are left alone.

For example:

```
#define quit(status)    quit(OK, !(status))
     quit(nerr == 0);
          /* expands to: quit(OK, !(nerr == 0)) */
```

The macro **quit** rewrites the argument list before calling the actual function **quit**.

Constructing String Constants

You can construct a string constant from an actual argument to a macro. For example:

```
#define STR(x) #x
     STR(f(1,2) + z)     /* expands to: "f(1,2) + z" */
     STR("Help!" he said.) /* expands to: "\"Help!\" he said." */
     STR(a/* commentary */b) * expands to: "a b" */
```

You determine the characters of the string constant by how you "spell" each of the tokens, and separating whitespace, in the actual argument. The preprocessor omits any leading and trailing whitespace. It replaces a comment with a single space. Other

compilers may replace any sequence of one or more whitespace characters by a single space, so you should write only actual arguments that have no space or a single space between tokens, if you want your program to be highly portable.

The preprocessor constructs the string from the actual argument, *not its expansion*. For example:

```
#define status (*getstat())
#define SHOW (x)   printf(#x " = %i\n", x)

     SHOW(status)        /* expands to:
          printf("status" " = %i\n", (*getstat())) */
```

Since adjacent string constants are concatenated, as described in Chapter 2, "Elements of the C Language," this becomes:

```
     printf("status = %i\n", (*getstat()));
```

Replacing Arguments Inside String Constants

As a compiler option, you can direct the preprocessor to replace argument identifiers *within* string constants when it expands a definition. For example:

```
#define SHOW(x) printf("x = %i\n", x)
SHOW(abc) /* expands to:
     printf ("abc = %i\n", abc)
```

Only string constants you write inside macro expansions are expanded this way.

You use this option to compile large programs written for a compiler that constructs strings this way, until you can alter the macro definitions to use the form shown above. Using **#x** to construct a string constant from an actual argument is a feature added to C in the proposed ANSI standard.

Predefined Macros

Two macros are predefined at the start of each compilation. You may not redefine either, test if either is defined, or remove either definition. They are:

The predefined macro **__LINE__** expands to a decimal integer constant whose value is the stored line number for the current line of source text. The first line number for any source file has the value 1.

The predefined macro **__FILE__** expands to a string constant whose value is the stored file name for the current source file. For example:

```
#define ERR(x) \
     printf(#x ": at line %i in %s\n", __LINE__, __FILE__)

if (MACH_TYPE != 370)
     ERR(unknown machine);
```

You can predefine additional macros when you invoke the compiler. Your *C Compiler User's Guide* describes how you specify macro definitions as command options to the compiler. Macros defined this way behave like any other macros you define. You may redefine them with exactly the same definition text, test if they are defined, and remove their definitions.

Constructing Tokens

You can construct a token from an actual argument to a macro. For example:

```
#define TMP(n)      tmp_var_ ## n

double TMP(1), TMP(2);  /* expands to:
                           double tmp_var_1, tmp_var_2; */
```

The operator **##** constructs a token from the operand tokens before and after it in a macro expansion. The preprocessor performs token construction after all other expansion has occurred. It "spells" the resulting token by concatenating the spellings of the expanded tokens before and after the **##** operator. All of the characters must contribute to the spelling of the resulting token. If the resulting token is a macro name, it is not expanded. If it is a **##** operator, it does not cause further token construction.

Removing Macro Definitions

The preprocessor directive

#undef *ident*

removes any definition of *ident* as a macro. It is not an error to remove a macro definition from an identifier not currently defined as a macro. If you wish to alter the definition of a macro, you must first **#undef** its identifier before you may **#define** it a different way.

You use the **#undef** preprocessor directive to remove any unwanted macro definition that may be present in a header file, as in:

```
#include <stdio.h>
#undef getchar      /* getchar no longer a macro */
    .....
    get_or_put(&getchar, &c);    /* getchar must be a function */
```

This ensures that your program contains no macro definition that masks the underlying declaration of **getchar** in the header file **<stdio.h>**. You can take the address of a function, but not of a macro. Chapter 8, "C Runtime Environment," discusses this use of the **#undef** preprocessor directive.

Information Preprocessor Directives

Several preprocessor directives communicate special information to the compiler. This section describes these directives.

Comments

The preprocessor directive

#

does nothing. It may contain only whitespace. You use such directives to hold comments, as in:

```
#if MACH_TYPE == 370
#
#    /* 370 version follows */
#
```

Line Control

The preprocessor directive

#line *const*

alters the stored line number to the value of the decimal integer constant *const*. The next line of source text has the new line number. The compiler displays the stored line number when it emits an error message. You can expand the stored line number into your program text by writing the predefined macro name **__LINE__**, as described earlier in this chapter. You use the **#line** preprocessor directive to correct for text lines added or deleted by programs that modify your source text, or translate other languages into C. For example:

```
#line 87
```

The preprocessor directive

#line *const* **"***fname***"**

also alters the stored file name to *fname*. The next line of source text has the new line number and the new file name. The compiler displays the stored file name when it emits an error message. You can expand the stored file name into your program text by writing the predefined macro name **__FILE__**, as described earlier in this chapter. You use this form of the **#line** preprocessor directive to replace the name of an intermediate file with that of a source file before it has been modified or translated. For example:

```
#line 87 "matrix.p"
```

Pragmas

The preprocessor directive

#pragma linkage(*ident***, OS)**

declares the identifier *ident* as a function with external linkage and the standard OS calling sequence. You use it to declare *ident* as a function written in a language that is called with the OS calling sequence. If you must declare additional information about *ident*, such as function return type and the number and types of its arguments, use a conventional declaration.

On a function call to *ident* your program uses the OS calling sequence, instead of the calling sequence used within C. Your *C Compiler User's Guide* describes the C calling sequences.

You can take the address of the function with the OS calling sequence, but you cannot assign it to any function pointer you can declare. Its type is unique. If you type cast it to another function pointer type, do not use the converted value to call the function. Its calling sequence will be incorrect.

Different compilers define other formats for the **#pragma** preprocessor directive. The compiler will ignore any format that it does not recognize. If you miswrite a **#pragma linkage** preprocessor directive, therefore, the compiler may not emit an error message. There are no formats defined for all compilers. Use the **#pragma** preprocessing directive only in programs you do not intend to be portable.

For information on mixing languages, see Chapter 8, "Interfaces with Other Languages," in your *C Compiler User's Guide*.

Chapter 8: C Runtime Environment

You build an executable C program file by typing a command that invokes **CC** and names all the files that make up the program. **CC** compiles any source files you name to produce object code files. You then invoke the standard IBM linkage editor or loader to link your object code files with the object code files you need from the C library. Chapter 3 of your *C Compiler User's Guide* describes how you invoke **CC**. You invoke the resulting C program file by typing another command that names the C program file and any command options you wish to specify to it.

Your C program executes in a "runtime environment" that provides access to command options, performs input/output, and returns control to the system when your program terminates. This chapter describes what you need to know about the runtime environment for C programs.

C Library Functions

There are no statements in C such as READ, WRITE, or OPEN, as there are in many other languages. Instead, you perform calls on C library functions. The C library performs input/output, dynamic storage allocation, string manipulation, and many other useful actions. A library function will typically:

* Take certain arguments as input parameters
* Perform a single operation
* Return a result as the value of the function, modify the values stored in data objects, or alter the state of a file.

Chapter 9, "Input/Output," describes the functions that perform input and output for your program. Chapter 11, "C Library Reference," describes *all* library functions.

The C library provides "header files" for use with the **#include** preprocessor directive. Header files make it easy for you to declare groups of related library functions and any types or data objects that go with them. For example, you can write text strings from C by using the code:

```
#include <stdio.h>
     .....
     printf("Second quarter report:\n");
```

In this example, **<stdio.h>** is the header file you include that defines the **printf** library function. **printf** writes the text in your output file. The library header files define many library functions as macros, by using the **#define** preprocessor directive. Such macros often expand to code that executes much faster than an actual function call, possibly at the cost of greater code size. If the library defines such a macro for a function, its definition will contain enough parentheses so that C requires no additional parentheses around the "function call" itself or around any of its arguments. The compiler evaluates each argument to such a macro exactly once, unless the description of the particular library function provides an explicit exception to this rule. This ensures that arguments with side effects, such as incrementing a data object, do not cause unpredictable or surprising results.

For example, you can write code such as:

```
#include <stdio.h>
     .....
     output_char = putchar(*p++) + 512;
```

regardless of whether or not **<stdio.h>** defines **putchar** as a macro.

Since the notation for writing a macro expression with arguments is the same as for calling a function in C, it does not matter whether or not the header file defines a macro for a function. It does matter, however, if you wish to take the address of a library function by writing the operator **&** before its name, as in **&putchar**. If **putchar** is defined as a macro, the compiler may generate an error message when you write the expression **&putchar**, or just **putchar**. If your program must refer to the actual library function instead of its macro replacement, you have two choices:

* Do not use the header file for that function, but declare it in your program just as it appears in its description in this manual
* Include the header file for that function, then remove any potential macro definition by using the **#undef** preprocessor directive

For example, your code may always take the form:

```
#include <stdio.h>
#undef putchar
     .....
     ptr_fun = &putchar;
```

since it is permissible to **#undef** an identifier not currently defined as a macro.

Of the two choices, you will probably find that using **#undef** is more convenient.

Program Startup and Termination

When you type a command that names a C program file, the system loads the program into memory and starts program execution. After some preparation, the C runtime environment calls the function **main**, which you must provide somewhere among your C source files. The time at which your program calls **main** is "program startup." Eventually, your program will either return from **main**, call the library function **exit**, or call the library function **abort**. The time at which one of these events occurs is "program termination." After some preparation, the C runtime environment returns control to the system and the execution of the C program file is complete.

Before program startup, the system copies command options you specify into alterable memory and terminates each with a null character. The first argument to **main** is an *int* giving the number of command options you specify plus one. The second argument to **main** is of type *array of pointer to char*. It points to an array of pointers to each of the command options. Each command option is a "string," which is a data object of type *array of char* whose last character is the null character.

For example, if you define **main** as:

```
int main(int argc, char *argv[])
        {.....}
```

and if the command options you specify are **"hello"** and **"world"**, then upon entry to **main**

* **argc** has the value 3
* **argv[1]** points at the string **"hello"**
* **argv[2]** points at the string **"world"**
* **argv[3]** is a null pointer

The array **argv** always has at least two members. The first argument, **argv[0]** is always defined, and there is always a null pointer stored in **argv** just beyond the last command option. Moreover, **argc** is always greater than zero upon entry to **main**. If the C runtime environment can determine the name of the program you specify with your **CC** command, then **argv[0]** points at a string giving the program name. Otherwise, **argv[0]** points at the string **""**. Your *C Compiler User's Guide* describes the value assigned to **argv[0]** on your system.

You can write a command option containing spaces by enclosing the spaces within double quotation marks. For example, **"c d e"** is a single command option containing two spaces, not three separate command options. You write a double quotation mark within a command option as **\"** and a backslash character as ****. Note that other escape sequences and trigraphs are *not* recognized inside command options.

Under TSO, the runtime environment translates the command line to lowercase. Any characters which you do not want the runtime

environment to translate can be escaped by preceding them with a backslash character.

Under IMS, there are two ways to invoke a C program. If the C program is invoked directly from IMS, **argc** is set to 1, and **argv[0]** is set to **""**. If the C program is invoked from the **system** function, **argc** and **argv** are initialized from the command string passed to the **system** function.

Note that under IMS signals are not set to their default values when IMS or the **system** function calls **main**. If you alter signal handling by calling **signal**, you may affect the entire IMS environment.

Before program startup, the C runtime environment sets up the data object **_pcblist** for IMS programs. **_pcblist** is available to all user programs, whether or not they are running under IMS. For more information on **_pcblist**, see the description of the **ims.h** header file in Chapter 11, "C Library Reference."

The program may alter the stored values of any of the characters in any of its command options.

Before program startup, the C runtime environment opens three text streams. The files are under control of three data objects that the C library provides. All three data objects are of type **FILE**. If you include the header file **<stdio.h>** in a C source file, your program will have the definition of the type **FILE** and definitions for macros that refer to these data objects. These macros are:

* **stdin**, controlling a readable text stream called the "standard input" stream
* **stdout**, controlling a writable text stream called the "standard output" stream
* **stderr**, controlling a writable text stream called the "standard error" stream

By convention, **stdin** is read by programs that process a sequential stream of input, where the name of the input file is of no importance to the program. Similarly, **stdout** is written by programs that produce a sequential stream of output, where the name of the output file is of no importance to the program. Error messages are written to **stderr**.

By default, standard output is displayed on your interactive terminal. See your *C Compiler User's Guide MVS, MVS/XA* (SC09-1129) for information on how **stdin**, **stdout**, and **stderr** are set up under MVS batch, where there is no interactive terminal. You can direct standard output to a named file by writing a command option that begins with **>**. The remainder of the argument is taken as the name of a file to create and open for text output. You can also append standard output to the end of an existing file by writing a command option that begins with **>>**. The description of the **fopen** function in Chapter 11, "C Library Reference" describes the rules for writing file names on your system. By redirecting standard output this way, you can

save the output of your program in the file of your choice.

Similarly, standard input by default reads what you type at your keyboard. You can direct that standard input come instead from a named file by writing a command option that begins with **<**. The remainder of the command option is taken as the name of a file to open for text input. By redirecting standard input this way, you can provide your program with input that you set up in advance.

Standard error output is displayed on your interactive terminal. If standard output is also left directed to your terminal display, then the display combines the two output streams. However, if you redirect standard output to a file, text written to standard error still appears on the display. You may not redirect standard error output on the command line.

If you specify the **RENT** option to **CC**, the table **C#INIT1** is used to initialize global and static data objects upon program startup. If you do not specify the **RENT** option, a dummy table in the C library is used.

Upon program termination, the runtime environment closes standard input, standard output, and standard error. It also closes any other files that the C library opened, so you do not have to close each file that you open within your program.

How Library Functions Indicate Errors

During execution, your program may call library functions in many different ways. If a library function cannot perform the service you request, an error occurs. The C library handles errors in a variety of different ways depending on their severity.

The simplest situation is where the error is a special case, one that occurs frequently. Here, the usual response is to return a unique value that your program can easily distinguish from any other value that that function might return. An example is the library function **strpbrk(s1, s2)**, which locates the first occurrence of any character from the string **s2** within the string **s1** and returns a pointer to that occurrence. If there is no such occurrence, **strpbrk** returns a null pointer.

A library function performing input from a file may encounter "end of file" while searching for additional input. Such a function may report this situation by returning a special value, such as **EOF**. The macro **EOF** is defined in the header file **<stdio.h>**. However, it will also record having encountered the end of the file by setting a field within the **FILE** structure it uses to control input from that file. Other functions in the library can later test and clear that indicator.

For example:

```
int ch;

while ((ch = getchar()) != EOF)
     putchar(ch);  /* copy characters until end of file */
if (feof(stdin))
     .....         /* control should come here */
```

A function performing input or output from a file may encounter an error from which it cannot recover when making requests to the underlying operating system. Such a function may report this situation by returning some indication of early termination. It will also record the read/write error by setting a field within the **FILE** structure it uses to control input/output to that file. Other functions in the library can also test and clear that indicator as they can the end of file indicator.

Many math functions are defined for only a subrange of all the input values that your program can specify to them. If you call a math function with an argument for which the function is not defined, a "domain error" occurs. An example is the square root of a negative value, as in **sqrt(-1.5)**. Also, your program can call a math function with an input value that yields an output value too large or too small for the computer to represent. This is a "range error."

When either a domain error or a range error occurs within a library function, the library records the occurrence in a static data object within the library. The header file **<stdefs.h>** defines **errno** as a macro that refers to this static data object, which is of type *int*. Your program may set **errno** to zero, call a library function, and then test whether **errno** has been set to some nonzero value, indicating that an error occurred. For example:

```
errno = 0;
y = alpha * exp(x * x / sigma);
if (errno)
     printf("y overflowed\n");
```

Like the functions in the library, you should not try to take the address of **errno**. The header file **<stdefs.h>** also defines the macros **EDOM** and **ERANGE**. **EDOM** is the value that a function stores in **errno** when a domain error occurs, and **ERANGE** is the value that a function writes to **errno** when a range error occurs.

The description of each library function tells whether it sets **errno** and what its return value is on various errors. A macro called **HUGE_VAL**, which **<math.h>** defines, is a value of type *double* that a function returns in place of a value too big to represent. If a value is negative and too large to represent, the function returns **-HUGE_VAL** instead. These are both instances of "floating overflow." Your program should not compare values against **HUGE_VAL** to check for floating overflow. Testing **errno** is better programming practice and is more portable as well.

If a value is too close to zero to represent, the function replaces it with the exact value 0. This is "floating underflow." Both overflow and underflow are range errors.

The most extreme response to an error that a library function detects is to write an error message to the standard error stream and then terminate abnormally. This happens, for example, when the C runtime attempts to redirect standard output, as described above, to a file it cannot create. Appendix B, "Runtime Error Messages," documents all such cases.

How Your Program Indicates Errors

You specify a program termination status as the argument to **exit** or the return value from **main**. In either case, the value 0 specifies "successful termination." All other values specify "unsuccessful termination."

You can also cause an abnormal termination by calling **abort**. The function **kill** calls **abort** if default signal handling is specified for the signal it is reporting.

Your program can regain control from a call to **abort**. You must specify a function to call when the "signal" **SIGABRT** is reported, as in:

```
#include <setjmp.h>
#include <stdlib.h>
#include <signal.h>
....
static jmp_buf ckpoint;

/*   handle abort signals
 */
void restart()
     {
     printf("abort signal occurred\n");
     longjmp(ckpoint, 1);
     }
     .....
     setjmp(ckpoint);    /* restart here */
     signal(SIGABRT, &restart);
     process();
```

Once your program calls the function **signal**, subsequent occurrences of the signal **SIGABRT** will be handled by calling **restart**. The signal handler **restart** prints an error message, then calls **longjmp** to resume control just after the call to **signal**. Normally, your program should not handle the signal **SIGABRT**. If it must keep running at all costs, however, your program can regain control as shown here.

Chapter 11, "C Library Reference," describes all of the functions shown in this example.

Chapter 9: Input/Output

You perform input/output in C by calling C library functions. You can choose among three groups of functions:

* Formatted input/output permits easy conversion of printable text input to internal representation, or internal representation to printable text output.
* Uninterpreted text input/output lets you do your own formatting.
* Binary input/output permits transparent copying of data objects to and from storage.

In all cases, the C library buffers input/output for enhanced performance.

You include the header file **<stdio.h>** to declare the input/output functions, several macros, and the type **FILE**.

Streams

All the input/output functions in the C library convert input and output to logical "streams" of data. Streams have uniform properties whether you are writing to or reading from physical devices or files on storage media. Each open file appears as an ordered sequence, or "stream," of eight bit bytes. The C library makes available to your program no other structure or control information.

There are two forms of streams: text streams and binary streams. A text stream is an ordered sequence of characters made up of text "lines." Each text line consists of zero or more characters plus an end of line indicator. Inside your C program, the character newline **'\n'** indicates the end of each line, regardless of how the end of a line is indicated in the external file. There is not a one to one correspondence between the characters in a text stream and the characters in the file. Unless you write only the "printable" characters, data you read in from a text stream may not compare equal to the data you wrote out to the same text stream earlier. The C library may lose or alter an incomplete last line on output. However, the last character you read in from a text stream will always be the last character that you wrote out to the text stream earlier, as long as that last character is a newline.

A binary stream is an ordered sequence of characters that represents internal data exactly as written. Underlying record structure is not visible to your program when it reads a binary stream. Data you read in from a binary stream always compares equal to the data you wrote out to that binary stream earlier. The C library may, however, append one or more null characters `'\0'` to the end of some binary files that your program writes.

Files may have either fixed record format or variable length record format. The default is variable length record format for text files and fixed record format for binary files. You can specify whether a file is in fixed record format or variable length record format when you open it. Files of fixed record format contain records all of the length you specify. Files of variable length record format may contain records of any length up to the maximum length you specify. If you do not specify a record length for a fixed record format file when you create it, the default record length is 80 characters. If you do not specify a record length for a variable length record format file when you create it, the default maximum record length is 255 characters.

The description of the function **fopen** in Chapter 11, "C Library Reference," tells you how to determine what the file format and record length will be when you open a file.

Text Streams

Note the following special considerations when you read or write text streams under MVS and VM/CMS:

* When you write variable length record format files, the C library converts all "empty" lines to lines containing a single space character. The C library converts the internal sequence "newline newline" to "newline space newline" when it writes it to a file. When the C library reads a variable length record format file, it interprets a record containing a single space character as an "empty" line.

* If your program *replaces* a record within a variable length record format file, it will fail if it attempts to write data that alters existing record boundaries. Your program may replace a record with one of the same length, but the write operation will fail if the size of the new record differs from the size of the old record. If your program *appends* a record by writing a line of text that is longer than the record size established for the file, the write operation will fail.

* Records you write to fixed record format text files are padded with spaces. Trailing spaces are removed when you read a fixed record format file.

* If your program *replaces* a record within a fixed record format file by writing a line of text that is longer than the record size, the write operation will fail. If your program *appends* a record by writing a line of text that is longer than the record size, the extra characters are discarded and

the write operation does not fail.

* You cannot create an empty text file. If you close an output text stream without writing any data to it, the C library creates no external file in conjunction with the text stream. For example, the sequence

```
FILE *f;

f = fopen("x.y.z", "w");
fclose(f);
```

will not generate an empty disk file. The file **x.y.z** does not exist after the sequence is executed.

* The return value of the **ftell** function is an encoded value that contains sufficient information for the C library to return to the file position at a later time during program execution, by calling the function **fseek**. You cannot perform arithmetic upon the encoded value. Chapter 11, "C Library Reference," contains more information on restrictions on **fseek** and **ftell**.

* When your program writes datasets with the formats A or M under MVS and MVS/XA, the characters newline **'\n'**, carriage return **'\r'**, and form feed **'\f'** are all converted to their corresponding ASA control characters on output.

* When writing to interactive terminals and printer devices, the character horizontal tab **'\t'** expands to between one and eight spaces. When your program writes to an interactive terminal, the character alert **'\a'** does not generate an audible alarm.

Binary Streams

Note the following special considerations when you read or write binary streams under MVS and VM/CMS:

* For fixed record format binary files, the C library packs the stream of characters you write into complete records. It pads an incomplete last record with null characters **'\0'** to achieve the proper record length.

* For variable length record format files, you can use the function **fflush** to create binary files with a specific record structure. To terminate the current record being appended to the file, perform a call on **fflush** for the stream. You also terminate the current record being appended if the data being written exceeds the record length for a given stream. For example, if you specify the maximum record length of a binary file to be 255 characters, and you attempt to write a block of 510 characters, the C library creates two 255 character records.

* The return value of the **ftell** function contains the number of characters from the start of the file when you call it for a binary stream connected to a fixed record format file.

For variable length record format files, **ftell** returns an encoded value that contains sufficient information for the C library to return to the file position at a later time during program execution, by calling the function **ftell**. Chapter 11, "C Library Reference," contains more information on restrictions on **fseek** and **ftell**.

* On MVS and MVS/XA, the C library regards a U-format dataset as a sequence of characters. The C library preserves the existing block structure of a U-format file when the file is open for updating.
* On MVS and MVS/XA, the C library regards the control character in datasets with formats containing an "A" (FA, FBA, etc.) as an extra character in each record.
* The C library does not support D-format datasets.

Buffered Input/Output

The C library functions declared in the header file **<stdio.h>** support buffered input and output. Buffering input and output helps reduce operating system overhead by reading or writing up to **BUFSIZ** characters on each "read" or "write" system call. **BUFSIZ** is a macro defined in the header file **<stdio.h>** as the default size in characters of the buffer that a stream uses. The functions in **<stdio.h>** perform buffered input/output by manipulating structures of type **FILE**. **FILE** is a data object type declared in the header file **<stdio.h>**. A **FILE** structure holds all the information necessary to control a stream, such as:

* A pointer to the current character in the file that the stream is manipulating
* A pointer to the buffer associated with the stream
* An indicator that records whether a read/write error has occurred
* An indicator that records whether the end of the file has been reached.

The C library performs the following basic operations on files and streams:

open a file: Establish a connection between a stream in your program and a data file on disk, your interactive terminal, or some other device.

close a file: Break a connection that you set up previously between a stream and a file.

read from a stream: Transfer characters from a stream into a buffer, given the address of the buffer within your program and the number of characters to read.

write to a stream: Transfer characters from a buffer to a stream, given the address of the buffer and the number of characters to write.

position a stream: Move to a given position within the file and continue reading or writing at that point.

change the buffering mechanism: Use a stream buffer you declare instead of an internal buffer that the C library sets up for you, or change the buffering strategy.

handle errors: Check for end of file or the occurrence of an unrecoverable read/write error.

Opening Files

You associate an input or output stream with a file by "opening" or "creating" the file. The file may be an existing text or binary file or a physical device. Creating a file opens a file of zero length. If you create a file that already exists, you truncate it. You cause its contents to be discarded, so that it appears as a file that did not previously exist.

To open a file, call the function **fopen**. You pass **fopen** a pointer to a "string" containing the name of the file and a pointer to a string that specifies how you want to manipulate the file. If it succeeds, the function returns a pointer to a **FILE** structure that controls the stream connected to the file. A string is a data object of type *array of char* whose last character is a null character. The description of the function **fopen** discusses how files are named on your system. You can manipulate a stream as either text or binary, for reading, writing, or updating. You specify these choices in the second argument, plus additional information such as record length. Some simple examples are:

```
FILE *pbfi, *pbfo, *ptfi, *ptfo;

if (!(ptfi = fopen(av[1], "r")) ||     /* reading text */
    !(ptfo = fopen(av[2], "w")))       /* writing text */
    printf("cannot open text files\n");

if (!(pbfi = fopen(av[3], "rb")) ||    /* reading binary */
    !(pbfo = fopen(av[4], "wb")))      /* writing binary */
    printf("cannot open binary files\n");
```

You could write an explicit file name, such as **"data.in"**, as the first argument to **fopen**. In practice, however, you should avoid writing explicit file names into your source text. Your program can obtain file names from the command options, as input text from another file, or in response to an interactive query. Using any of these methods makes your program more flexible.

You can also open a file by calling **freopen**, which reuses an existing **FILE** data object, such as **stdin**. When you call **freopen**, the function closes any open file controlled by the specified **FILE** data object, then opens the file you name, just as with **fopen**. Chapter 11, "C Library Reference," discusses all of the options for opening a file, with the description of the function **fopen**.

The runtime environment provides you with three open text files at program startup: the "standard input" stream **stdin**, for reading input to your program; the "standard output" stream **stdout**, for

writing output from your program; and the "standard error" stream **stderr**, for writing error messages from your program. Chapter 8, "C Runtime Environment," provides more information on the standard streams. In many cases, your program need not call **fopen** or **freopen** because you can perform all necessary input and output using the three standard streams. If your input is a stream of text that your program reads just once, from start to finish, and if your output is a stream of text that your program writes sequentially, then use the standard streams. Your program will be more general if you do, and you will be able to redirect the input and output each time you invoke the program.

Closing Files

You disassociate a file from a stream by closing the file. This automatically "flushes" the output stream. The C library writes any remaining buffer contents to an output file. When your program terminates, the C library automatically closes all the files your program has opened. Your program should still close each open file, however, as soon as all operations on it are complete. You are less likely to encounter any limits on the number of open files if you close them promptly. In addition, an unexpected program termination, such as from a system failure, is less likely to corrupt the contents of your files if they are closed.

To close a file, call the **fclose** function. You pass **fclose** the pointer to the **FILE** data object that **fopen** returns.

The following example shows how you call **fopen** and **fclose** in combination to open and close files in a C program.

```
#include <stdio.h>

int main(ac, av)
     int ac;
     char *av[];
     {
     FILE *pf;

     for (; 0 < --ac; ++av)  /* for each command option */
          if (!(pf = fopen(av[1], "rb,recfm=f,lrecl=80")))
               printf("can't open %s\n", av[1]);
          else
               {
               process(pf);   /* go process file */
               fclose(pf);
               }
     exit(0);
     }
```

In this example, the program must call **fclose** so that the number of open files does not increase steadily. The example assumes that some separately compiled function **process** processes each opened file.

Reading Streams

The C library provides several different functions for reading characters from a stream:

* You can read a character at a time from the standard input by calling **getchar**.
* You can read a character at a time from a stream you specify by calling **getc**.
* You can read a line of text into an array you specify from a stream you specify by calling **fgets**.
* You can read binary data into an array you specify from a stream you specify by calling **fread**.

For example, you can copy the standard input to the standard output by writing:

```
int ch;

while ((ch = getchar()) != EOF)
    putchar(ch);
```

The header file **<stdio.h>** may redefine any of these functions as a macro. The macro version of **getchar**, for example, obtains the next input character directly from the buffer in **stdin**, if there is one. It calls the actual library function only if the buffer is empty.

Using the macro version of an input or output function often substantially improves the performance of your program. Be careful, however, not to write an argument to **getc** that has side effects, such as **getc(*ppf++)**. The macro version expands to an expression that evaluates its argument more than once. This is one of the two cases where the C library does not make the replacement of functions by macros transparent for function calls. The other case, the macro **putc**, is described later in this chapter.

You can "push back" a character you have read from an input stream, by calling the function **ungetc**. This is useful in programs that must "parse" or "recognize" input. Often such a program must read characters until it finds one it cannot currently process. Your program is simpler if it can just push the unwanted character back, and have it delivered as the next character read. The **ungetc** function performs this service.

Chapter 11, "C Library Reference," documents all of these functions. It also documents two other functions that read buffered input: **fgetc** and **gets**. These are older functions whose capabilities are better provided by the functions mentioned above. Do not use **fgetc** or **gets**.

Writing Streams

The C library provides several different functions for writing characters to a stream:

* You can write a character at a time to the standard output by calling **putchar**.

* You can write a character at a time to a stream you specify by calling **putc**.

* You can write a text string to the standard output by calling **puts**.

* You can write a text string to a stream you specify by calling **fputs**.

* You can write binary data from an array you specify to a stream you specify by calling **fwrite**.

The header file **<stdio.h>** may redefine any of these functions as a macro. The macro version of **putchar**, for example, stores the argument character directly into the buffer in **stdout**, if there is room. It only calls the actual library function if the buffer is full.

Be careful, however, not to write a file argument to **putc** that has side effects, such as **putc(c, *ppf++)**. The macro version expands to an expression that evaluates its argument more than once. This is the other case, besides **getc** described earlier in this chapter, where the C library does not make the replacement of functions by macros transparent for function calls.

You can ensure that the buffer for an output stream is written to the output file, by calling **fflush**. All output stream buffers are written upon program termination. Normally, you need to call **fflush** only to ensure that the C library display output to an interactive terminal before your program requests input. You can also alter the buffering strategy for a stream so that the C library flushes the buffer on every write, as described later in this chapter.

Chapter 11, "C Library Reference," documents all of these functions. It also documents one other function that writes buffered output: **fputc**. This is an older function whose capabilitities are provided by the functions mentioned above. It is included in the library for compatibility with older programs. Do not use **fputc**.

Positioning Streams

When you read or write a stream, the C library refers to a stored "file pointer" to determine which characters in the file to transmit. After it transmits the characters, the C library advances the file pointer by the number of characters transmitted.

You can obtain the current value of the file pointer for a stream by calling **ftell**. You can use this value later, while the file is still open, to return the file pointer to that place in the file, by calling **fseek**. You can position the file pointer at the beginning of the file by closing it and reopening it, by calling **fseek** or **rewind**. You can position the file pointer at the end of the file, by calling **fseek**. A binary file may have extra null characters appended to it, however, so the end of a binary file is not well defined.

For a binary stream connected to a fixed record format file, the value of its file pointer is the offset in characters from the start of the file to the next character to read or write. A value of zero indicates the first character of the file. You can perform integer arithmetic on a file pointer for such a binary stream, then use it in a later call to **fseek**. You can position the file pointer for such a binary stream at an explicit offset from the start of a file, or relative to the current file pointer, by calling **fseek**.

The value of a text stream file pointer is encoded in a special way, however. This is also true for a binary stream connected to a variable length record format file. Do *not* perform arithmetic on a file pointer for such a stream. If you use the value returned by **ftell** to reposition a file pointer for a text stream, do not write a text line that differs in length from the text line you wrote earlier.

Other Stream Services

The C library allocates storage for a stream buffer automatically when you first read or write the stream. Storage for the buffer is freed automatically when you close the file. You may, however, specify your own buffer, or specify an alternate buffering strategy, by calling **setvbuf**. The alternate buffering strategies include: full buffering, line buffering, and no buffering. An older version of this function is **setbuf**, which you should not use.

To check whether end of file has been encountered on a stream, call **feof**. To check for an irrecoverable transmission error, call **ferror**. To clear these indicators, call **clearerr**. Most functions that read from a stream provide a return value that indicates whether end of file or an error have occurred. Your program may have no need for any of these services. The section "How Library Functions Indicate Errors," in Chapter 8, "C Runtime Environment," discusses these and other error indications.

Formatted Input/Output

The C library provides a set of "scan" functions that convert printable text, usually from an input stream, to values that the functions store in data objects. The C library also provides a set of "print" functions that convert the values of expressions to printable text, usually for output to a stream. These are the "formatted" input/output functions. All are functions that accept a variable length argument list. All are functions that have a "format string" argument. The format string determines how many arguments are actually present on a given call. It also determines how each argument is to be converted. You write scan format strings by rules very similar to the rules for writing print format strings, even though the conversions go in different directions.

For example:

```
int xx;

scanf("xx=%i", &xx);          /* read an integer field */
printf("xx=%i", xx);          /* write an integer field */
```

The function **scanf** reads characters from the standard input under control of its format string argument. The format string in the example requires that the next input characters be **xx=**, followed by a sequence of characters that obey the rules for an integer constant. If the input meets these requirements, the function stores the converted value in the data object of type *int* pointed at by the argument **&xx**. The function **printf** writes characters to the standard output under control of its format string argument. The format string in the example generates the characters **xx=**, followed by the value of the expression in **xx** written as an integer constant.

The character **%** within a format string introduces a "conversion specification." The characters following the **%** tell what data object type the function must convert. They may also specify the number of characters to produce or process. These characters may qualify the conversion in several ways. The characters between conversion specifications have more literal meanings. These characters must match input, and they are written as they appear on output. The descriptions of **fscanf** and **fprintf** in Chapter 11, "C Library Reference," cover all aspects of conversion specifications and format strings.

Formatted Input

The C library provides three different functions for converting printable text to values to be stored in data objects:

* You can read the standard input, by calling **scanf**.
* You can read a stream you specify, by calling **fscanf**.
* You can read a string stored in memory, by calling **sscanf**.

In all cases, there is a format string argument that controls the conversions. You write additional arguments for each of the data objects you wish to store values in. All of these arguments must be data object pointers. The value returned by each of these functions is a count of the number of data objects stored into. This number can be zero if the function detects a conflict before the first data object conversion specification completes.

You can, for example, convert input until end of file by writing a simple loop:

```
while (scanf("%Lf", &x))
    printf("cos(%Lf) = %Lf\n", x, cos(x));
```

You use **scanf** as the most convenient way to obtain input values from the standard input. You use **fscanf** to specify a stream other than the standard input. You use **sscanf** to try several different formats on the same input, as in:

```
while (fgets(buf, sizeof buf, stdin)) /* for each line */
    for (i = 0; i < sizeof fmt / sizeof fmt[0]; ++i)
        if (sscanf(buf, fmt[i], &x, &y, &z) == 3)
            return (YES); /* converted all values */
return (NO);  /* failed */
```

This example reads the standard input one line at a time, using **fgets**. It then tries to convert the string stored in **buf** by calling **sscanf** with a series of format strings. Pointers to format strings are stored in the array **fmt**. The first format string that causes **sscanf** to store values in all three data objects causes the function to return with a value of **YES**. If no format string succeeds, the function returns the value **NO**.

Formatted Output

The C library provides several different functions for converting expressions to printable text. Three of these are used most often:

* You can write to the standard output, by calling **printf**.
* You can write to a stream you specify, by calling **fprintf**.
* You can store characters as a string in memory, by calling **sprintf**.

In all cases, there is a format string argument that controls the conversions. You write additional arguments for each of the expressions you wish to convert to printable text. The value returned by each of these functions is a count of the number of characters of printable text produced.

You use **printf** as the most convenient way to write messages to the standard output. You use **fprintf** to specify a stream other than the standard output. You use **sprintf** to convert values to printable text that your program will further manipulate as data objects.

There are three additional functions for converting expressions to printable text. These are: **vprintf**, **vfprintf**, and **vsprintf**. They permit you to design variations of the print functions, with different initial arguments, special processing upon function entry, or special processing before function return. You use them only for more sophisticated applications.

Chapter 10: Organizing Your Program

C imposes very few constraints on how you lay out your program, or how you organize it into source files. You must make your program readable, however, so that you and others can maintain it. This chapter describes some of the conventions that C programmers have found useful for organizing programs. You can use it as a starting point for developing your own style for organizing your programs.

File Layout

Many C programs fit in one source file. Lay out those that do in the following order:

```
/*   TITLE OF PROGRAM
 *   ownership
 *
 *   brief description
 */
#include <stdio.h>       /* library header files */
#include <stdlib.h>

#define MAX_ELEM   50        /* macro definitions */

typedef double Num;          /* type definitions */

Num add(Num x, Num y);       /* function prototypes */
Num store(Num *px, Num y);
Num sub(Num x, Num y);
```

```
int stack_top = MAX_ELEM;
    /* static lifetime data objects */
Num stack[MAX_ELEM];

/*   add two numbers          functions in alphabetical order
 */
Num add(Num x, Num y)
    {
    return (x + y);
    }

/*   MAIN
 */
int main(int ac, char **av)
    {
    .....
    }

    .....                      /* other functions */
```

The components, in order, are:

Title: Use a comment to name your program, note ownership, and briefly describe what the program does. Place any copyright notice here.

Library header files: Every library function has a header file that declares it, as well as any useful type definitions and macros that go with it. Include header files for every library function you use. You can include them in any order.

Macro definitions: You use macros to give mnemonic names to numeric values that you use throughout a program. Write these "manifest constants" in uppercase letters so they stand out from data object identifiers. You also write "in line functions" as macros with arguments. Keep them simple.

Type definitions: Define any types you use throughout your program, particularly ones you might want to change uniformly. Type definitions also give mnemonic names to standard types you use in special ways. Type definition names begin with an uppercase letter so they stand out from standard C types and from other identifiers.

Function prototypes: Declare all your functions before you use them, and declare their arguments. If all function calls are in scope of a function prototype, the compiler can check arguments for proper types and number, and can convert them to the proper type. The compiler also checks function return types. Write function names with lowercase letters. On System/370, the first eight characters of function names with external linkage must be distinct.

Static lifetime data objects: Declare all data objects that must have static lifetime before you use them. If only one function uses a data object, declare it within the function body. If

it must be shared among functions, declare it within the function body. Keep the number of static lifetime data objects to a minimum. You write data object names with lowercase letters. On System/370, the first seven characters of data object names with external linkage must be distinct.

Function definitions: Define all your functions at the end of the source file, in alphabetical order by function name. You may wish to put **main** at the top of the list. Use the combined function prototype and definition form of the function type attribute. It is most easily compared with the function prototype at the beginning of the source file. If you must write C programs that also compile with older compilers, however, use the form with separate argument level declarations. You can then just "comment out" the function prototype argument lists at the beginning, as in:

```
Num add(/* Num x, Num y */);  /* looks like Num add( ) */
```

to get the older form of function declaration.

The example shows the bare minimum of comments a program needs. Use comments as much as you think necessary to supplement understanding of the code.

Function Layout

Adopt a consistent style for writing function definitions and adhere to it rigorously. If you can set horizontal tab stops every four to eight columns, then you can more easily indent in a uniform manner. The following example shows a consistent style for indenting:

```
Type name(Type1 arg1, Type2 arg2)
     {
     register int *p;    /* register declarations first */
     Type1 a1, a2;       /* then autos */
     static char first;       /* then statics */

     if (test)           /* if statement */
          x = y + 2;
     if (x || y)         /* if/else  statement */
          y = 0;
     else
          {
          ++x;
          --y;
          }
     if (test1)               /* else/if chain */
          fun1(x);
     else if (test2)
          fun2(y);
     else
          fun3(z);
     switch (value)      /* switch statement */
```

```
        {
    case A:
    case B:
        p = "first event";
        break;
    case C:
        p = "second event";
        break;
    default:
        p = "no match";
        }
    while (test)        /* while statement */
        {
        for (i = 0; i < 100; ++i)    /* for statement */
            x += i;
        for (; i < 200; ++i)
            x += f(i);
        for (sum = 0; ; )
            {
            register int j = 3;

            printf("top of loop\n");
            if (test)
                break;
            test = g(test, j);
            }
        for (; ; )          /* infinite loop */
            {
            printf("well?\n");
            if (getans())
                break;
                process();
            }
        }
    do  {               /* do/while statement */
        x += y;
        } while (x <= y);
    return (value);         /* return statement */
    }
```

Note the following:

Keywords: Always write a single space after every keyword.

Declarations: You may place declarations after any left brace, but most occur at the beginning of the function body. Leave an empty line after the last declaration to set it off from the executable statements that follow. Group declarations by storage class, with `static` declarations last. Place a declaration with a data initializer in a declaration by itself.

Statement Layout: The indenting that the example illustrates displays control relationships clearly. For each level of control nesting, indent one horizontal tab stop more. Do not exceed five levels of nesting. Break deeply embedded loops out into separate functions.

Expressions: Use no whitespace between operand and unary operators or addressing operators. Write a single space on either side of binary arithmetic operators. If you must continue an expression on another line, break it at the space next to an operator, if possible. Indent continuations one horizontal tab stop further than the start of the statement. Use parentheses for operators whose grouping you do not know well.

switch Statements: Write a *break* statement at the end of each group of case labels. If control must fall through the next case label, say so in a comment. Always write the **default** label last.

for Statements: Rewrite

```
for (; test; )
```

as

```
while (test)
```

Place only loop related expressions in the *for* statement. Use the *break* statement to exit from the middle for a *for*, *while*, or *do/while* statement.

return Statement: If an expression is present, write parentheses around it. Unless the *return* statement is inside a void function, specify a value on every *return*.

Restrictions

The following restrictions on coding practice reduce errors:

goto Statement: Avoid the *goto* statement completely if you can. If you must use it, never transfer control into a block. Write each label alone on a line, indented one tab stop less than normal.

do/while Statement: Rewrite *do/while* statements as *while* statements, if possible. A loop that can execute zero times is more robust than one that always executes at least once.

continue Statement: Replace a *continue* statement with an *if/else* statement inside a loop whenever possible.

Relational Operators: Do not use the operators > and >=. Tests such as

```
if ('0' <= c && c <= '9')
```

show interval tests better.

Const Types: Do not declare a data object as alterable if you can declare it with a const type instead. The more restrictions you impose on how much your program can alter data objects, the less likely it is that some future change will alter them unexpectedly.

Programs with Multiple Files

Once your program exceeds 500 to 1,000 lines, break it into multiple source files. When you do, group related functions and data objects in the same file. You can then give many of them internal linkage, by declaring them with storage class **static** at file level. Minimize the number of identifiers with external linkage.

Group all common macro definitions, type definitions, and function prototypes into one or more header files of your own. Include them at the top of each source file on an "as needed" basis, using the form:

```
#include "myhdr.h"
```

Each header file should have a comment that describes its unifying principle. Header files can in turn include standard header files from the C library, but they should not include other files that relate only to your program. Do not write any declarations with definitions in a header file. If you do so for an identifier with external linkage, and then include the header in two or more compilations, multiply defined external identifier results.

You can use the same file layout for a multifile program as for a program consisting of a single file. As with header files, write a comment describing the criteria for placing declarations in this file. You often include more include files in a multifile program than in a program that consists of only one file. You probably use fewer macros and type definitions unique to a compilation. The storage class **static** is more often used in file level declarations. Otherwise, the format of multifile programs is very similar to the one shown earlier.

Portability Issues

You can write C programs that run on many different architectures and operating systems, but you must plan carefully to do so. Learn which aspects of C vary most among computers, then restrict your programs so they avoid unnecessary dependencies on a given environment.

This manual gives warnings throughout about the aspects of C that affect portability. Follow the suggestions provided and you will avoid the worst problems:

Identifiers: Some systems restrict external identifiers to as few as six significant characters, in one case.

Text Streams: If your program processes text streams, remember that some characters that do not print may disappear if you write them out and read them back later. You can only position a file pointer at the beginning or end of a text stream, or to a file position your program determined earlier by calling **ftell**. Do not write long text lines. Do not

write an incomplete last line to a text stream.

Binary Streams: The text streams **stdin** and **stdout** can corrupt a stream of binary data. Open all binary files by name, with proper attributes. Assume that the operating system adds some number of null characters to the end of a binary file. Choose binary file formats that are self defining in length. Do not position the file pointer at the end of a binary file.

Byte Order: Data objects that occupy multiple bytes of storage differ in byte order among different architectures. You cannot write a four byte integer, for example, to a binary file and assume that you can recover that integer value when you read that file on a different system. You can write such data to a binary file if you know it will be read on a machine of the same architecture. Otherwise, convert binary data to some text representation and write a text file. You can also put the bytes in a canonical order when you write them to the binary file. Then reorder the bytes properly for each machine that reads the file.

File Names: There are few file names that you can use unchanged on many different operating systems. Avoid writing file names into your programs. It is better to obtain all file names at runtime.

Integer Size: Remember that the size of an *int* data object varies from 16 to 32 bits among the most popular computers. If you know an integer must be large, declare it as *long*. If you know it must be small, declare it as *short*. On the other hand, use the type definitions **size_t** and **ptrdiff_t** to declare integers that must count all the bytes in the largest declarable data objects. In addition, *always* use the operator **sizeof** to determine the number of bytes occupied by a data object other than type *char*. Write expressions such as **~0** to set all one bits in an *int* value, not **0xffff** or **0xffffffff**.

Pointer Size: A data object pointer may occupy more bytes than an *int*. A data object pointer may differ in size from a function pointer. Do not treat all integers and pointers as having the same size. Function prototypes are the best way to prevent this programming practice.

Keep in mind that many C programs are now running unchanged on machines as diverse as the IBM/370 and the IBM PC. If you want to write portable programs, C is better than many languages at allowing you to do so.

Chapter 11: C Library Reference

This chapter describes all of the facilities provided by the C library. The descriptions are organized by "header files." Appendix C, "Summary of Reserved Identifiers," contains a summary of all identifiers reserved in C or defined in the header files.

Header Files

You include a header file, with the **#include** preprocessor directive, in a compilation if your program uses any of the facilities associated with that header file. You may include the C library header files in any order. A header file may do any or all of the following:

define types: Any special types you need are defined

define macros: A macro may provide a special value you may need, designate a data object you may need to access, or expand to a statement that you cannot express directly

declare functions: All the functions that relate to a header file are declared within the header file.

In addition, every function declared in a header file may also be "masked" by a macro of the same name. When you write an expression that calls the function, the macro is expanded instead. The macro may expand to an equivalent expression that avoids calling a function, or it may directly call a lower level function and avoid one level of function nesting. You can write calls to any library function without worrying about whether or not it is masked by a macro definition, unless the description of the function says not to.

Chapter 8, "C Runtime Environment," discusses the limitations of macros that mask C library functions. Chapter 9, "Input/Output," describes the C library functions that perform input and output. Chapter 10, "Organizing Your Program," suggests a style for using C library header files.

The header files are described in alphabetical order. They are:

assert.h – runtime assertion checks

ctype.h – character test functions

ims.h – IMS access

limits.h – environmental limits

math.h – mathematical functions

setjmp.h – nonlocal jumps

signal.h – exceptional condition handling

stdarg.h – walking argument lists

stdefs.h – common definitions

stdio.h – stream I/O

stdlib.h – general utilities

string.h – string functions

time.h – timekeeping functions

Following each header file description are descriptions of all the functions declared in the header file. The functions appear in alphabetical order.

NAME

assert.h – header file for runtime assertion checks

SYNOPSIS

```
#include <assert.h>
```

FUNCTION

The header file **<assert.h>** declares the macro **assert**. You use **assert** to place consistency checks throughout your program. Your program performs these checks as it executes, and emits an error message if any of the checks fail. By defining the macro **NDEBUG** before you include **<assert.h>**, you can suppress these checks.

<assert.h> defines the following macro:

assert – test an assertion. **assert** expands to a statement. It must be followed by a semicolon.

<assert.h> *refers to* the following macro:

NDEBUG – suppress assertion checks if defined.

NAME

assert - test an assertion

SYNOPSIS

```
#include <assert.h>
void assert(type test);
```

FUNCTION

The **assert** macro performs a consistency check or tests an assertion at runtime. It expands to a statement that compares the test expression *test* against zero. *type* must be a scalar type. If the expression compares equal to zero, the statement emits an error message and calls the function **abort**. The error message takes the form:

Assertion failed: *test* **file** *file*, **line** *line*

where *test* is the expression you write as the argument to **assert**, *file* is the name of the source file where you expand the macro, and *line* is the line number in that source file where you expand the macro. The expression is written to the standard error stream. **assert** then calls **abort** to terminate your program abnormally.

If the identifier **NDEBUG** is defined as a macro at the point in your compilation where you include **<assert.h>**, an expansion of the **assert** macro followed by a semicolon becomes a *null* statement.

Write **assert** as an *expression* statement containing a function call. The call must be followed by a semicolon.

RETURNS

Nothing. **assert** expands to a statement, not an expression.

EXAMPLE

To ensure that a subscript is in range:

```
assert(0 <= i && i <= sizeof a / sizeof a[0]);
```

SEE ALSO

<stdio.h>, **abort**

NAME

ctype.h – header file for character test functions

SYNOPSIS

```
#include <ctype.h>
```

FUNCTION

The header file **<ctype.h>** declares functions which test or change the classification of a character.

<ctype.h> declares the following functions:

isalnum – test for alphabetic or numeric character.

isalpha – test for alphabetic character.

iscntrl – test for control character.

isdigit – test for decimal digit.

isgraph – test for graphic character.

islower – test for lowercase character.

isprint – test for printable character.

ispunct – test for punctuation character.

isspace – test for whitespace character.

isupper – test for uppercase character.

isxdigit – test for hexadecimal digit.

tolower – convert uppercase character to lowercase.

toupper – convert lowercase character to uppercase.

All of these functions accept only a limited range of input values. Make sure that the value of the argument you pass can be represented as an *unsigned char*, or is equal to the macro **EOF**. These are the values returned by input routines such as **getchar** and **getc**. The header file **<stdio.h>** documents **EOF**, **getchar**, and **getc**.

SEE ALSO

<stdio.h>, **getc**, **getchar**

isalnum

NAME

isalnum – test for alphabetic or numeric character

SYNOPSIS

```
#include <ctype.h>
int isalnum(int c);
```

FUNCTION

isalnum tests whether *c* is an uppercase letter, a lowercase letter, or a decimal digit.

RETURNS

isalnum returns a nonzero value if the test is met.

EXAMPLE

To test for a valid C identifier:

```
if (isalpha(*s) || *s == '_')
     while (isalnum(*++s) || *s == '_')
          ;
```

SEE ALSO

isalpha, **isdigit**, **islower**, **isupper**, **isxdigit**, **tolower**, **toupper**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

NAME

isalpha - test for alphabetic character

SYNOPSIS

```
#include <ctype.h>
int isalpha(int c);
```

FUNCTION

isalpha tests whether *c* is an uppercase letter or a lowercase letter.

RETURNS

isalpha returns a nonzero value if the test is met.

EXAMPLE

To find the end points of an alphabetic string:

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

char *s = "   12345 bcdefghABCDEFhijkl #########";

main()
     {
     char *first, *last, *s1, *s2, outbuf[50];

     first = s;
     while (*first && !isalpha(*first))
          ++first;  /* set first to first alpha character */
     for (last = first; isalpha(*last); ++last)
          ;         /* set last to last alpha character */
     for (s1 = first, s2 = outbuf; s1 < last; ++s1, ++s2)
          *s2 = *s1;
                    /* copy alpha chars to output buffer */
     *s2++ = '\0';       /* terminate string with null */
     printf("original string: %s\n", s);
     printf("output string  : %s\n", outbuf);
     }
```

SEE ALSO

isalnum, **isdigit**, **islower**, **isupper**, **isxdigit**, **tolower**, **toupper**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

iscntrl

NAME

iscntrl – test for control character

SYNOPSIS

```
#include <ctype.h>
int iscntrl(int c);
```

FUNCTION

iscntrl tests whether *c* is a "control" character. A control character in EBCDIC is any character whose code is less than 0x40, except the codes 0x29, 0x30, 0x31, and 0x3E.

RETURNS

iscntrl returns a nonzero value if the test is met.

EXAMPLE

To map control characters to **%**:

```
#include <ctype.h>

char *mapstr(si)
     char *si;
     {
     char *s;
     for (s = si; *s; ++s)
          if (iscntrl(*s))
               *s = '%';
     return (si);
     }
```

SEE ALSO

isgraph, **isprint**, **ispunct**, **isspace**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

NAME

isdigit – test for decimal digit

SYNOPSIS

```
#include <ctype.h>
int isdigit(int c);
```

FUNCTION

isdigit tests whether *c* is a decimal digit.

RETURNS

isdigit returns a nonzero value if the test is met.

EXAMPLE

To convert a decimal digit string to its integer value:

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

main()
     {
     char *s1 = "534";
     char *s;
     unsigned int sum;

     for (sum = 0, s = s1; *s && isdigit(*s); ++s)
          sum = (sum * 10) + (*s - '0');
     printf("input string:  %s\n", s1);
     printf("output number: %d\n", sum);
     }
```

SEE ALSO

isalnum, **isalpha**, **islower**, **isupper**, **isxdigit**, **tolower**, **toupper**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

isgraph

NAME

isgraph – test for graphic character

SYNOPSIS

```
#include <ctype.h>
int isgraph(int c);
```

FUNCTION

isgraph tests whether *c* is a "graphic" character. A graphic character is any printable character except space.

RETURNS

isgraph returns a nonzero value if the test is met.

EXAMPLE

To output only graphic characters:

```
for (; *s; ++s)
     if (isgraph(*s))
          putchar(*s);
```

SEE ALSO

iscntrl, **isprint**, **ispunct**, **isspace**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

NAME

islower - test for lowercase character

SYNOPSIS

```
#include <ctype.h>
int islower(int c);
```

FUNCTION

islower tests whether *c* is a lowercase letter.

RETURNS

islower returns a nonzero value if the test is met.

EXAMPLE

To convert a string to uppercase:

```
for (; *s; ++s)
     if (islower(*s))
          *s += 'A' - 'a';    /* also see toupper */
```

SEE ALSO

isalnum, **isalpha**, **isdigit**, **isupper**, **isxdigit**, **tolower**, **toupper**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

isprint

NAME

isprint – test for printable character

SYNOPSIS

```
#include <ctype.h>
int isprint(int c);
```

FUNCTION

isprint tests whether *c* is a printable character.

RETURNS

isprint returns a nonzero value if the test is met.

EXAMPLE

To output only printable characters:

```
for (; *s; ++s)
     if (isprint(*s))
          putchar(*s);
```

SEE ALSO

iscntrl, **isgraph**, **ispunct**, **isspace**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

NAME

ispunct - test for punctuation character

SYNOPSIS

```
#include <ctype.h>
int ispunct(int c);
```

FUNCTION

ispunct tests whether *c* is a "punctuation" character. Punctuation characters include any printable character except space, decimal digits, lowercase letters, or uppercase letters.

RETURNS

ispunct returns a nonzero value if the test is met.

EXAMPLE

To extract all punctuation from a string into a separate buffer:

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

char *s1 = "abcdef !@#$%^&*()-_=+[{]}'\"mfnmdmsn\\|"~";

main()
     {
     char punct_buf[100];
     char *s;
     int i;

     s = s1;
     for (i = 0; *s; ++s)
          if (ispunct(*s))
               punct_buf[i++] = *s;
     punct_buf[i] = '\0'; /* terminate string with null */
     printf("input string: %s\n", s1);
     printf("output      : %s\n", punct_buf);
     }
```

SEE ALSO

iscntrl, **isgraph**, **isprint**, **isspace**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

isspace

NAME

isspace – test for whitespace character

SYNOPSIS

```
#include <ctype.h>
int isspace(int c);
```

FUNCTION

isspace tests whether *c* is a whitespace character. Whitespace characters are horizontal tab `'\t'`, newline `'\n'`, vertical tab `'\v'`, form feed `'\f'`, carriage return `'\r'`, and space.

RETURNS

isspace returns a nonzero value if the test is met.

EXAMPLE

To skip leading whitespace:

```
while (isspace(*s))
    ++s;
```

SEE ALSO

iscntrl, **isgraph**, **isprint**, **ispunct**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

NAME

isupper – test for uppercase character

SYNOPSIS

```
#include <ctype.h>
int isupper(int c);
```

FUNCTION

isupper tests whether *c* is an uppercase letter.

RETURNS

isupper returns a nonzero value if the test is met.

EXAMPLE

To convert a string to lowercase:

```
for (; *s; ++s)
     if (isupper(*s))
          *s = tolower(*s);  /* also see tolower */
```

SEE ALSO

isalnum, **isalpha**, **isdigit**, **islower**, **isxdigit**, **tolower**, **toupper**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

isxdigit

NAME

isxdigit - test for hexadecimal digit

SYNOPSIS

```
#include <ctype.h>
int isxdigit(int c);
```

FUNCTION

isxdigit tests whether *c* is a hexadecimal digit. The hexadecimal digits include the characters **0123456789abcdefABCDEF**.

RETURNS

isxdigit returns a nonzero value if the test is met.

EXAMPLE

To convert a hexadecimal digit string to its integer value:

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

char *s1 = "0Cf1";

main()
     {
     char *s;
     unsigned int sum, digit;

     for (sum = 0, s = s1; isxdigit(*s); ++s)
          {
          if (isdigit(*s))
               digit = *s - '0';
          else if (islower(*s))
               digit = (*s - 'a') + 10;
          else
               digit = (*s - 'A') + 10;
          sum = (sum << 4) + digit;
          }
     printf("input string: %s\n", s1);
     printf("output digit: %u\n", sum);
     }
```

SEE ALSO

isalnum, **isalpha**, **isdigit**, **islower**, **isupper**, **tolower**, **toupper**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

NAME

tolower – convert uppercase character to lowercase

SYNOPSIS

```
#include <ctype.h>
int tolower(int c);
```

FUNCTION

tolower converts an uppercase letter *c* to its lowercase equivalent. All other characters are left unchanged.

RETURNS

tolower returns the corresponding lowercase letter, or the unchanged character.

EXAMPLE

To convert a string to lowercase:

```
for (; *s; ++s)
     *s = tolower(*s);
```

SEE ALSO

islower, **isupper**, **toupper**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

toupper

NAME

toupper – convert lowercase character to uppercase

SYNOPSIS

```
#include <ctype.h>
int toupper(int c);
```

FUNCTION

toupper converts a lowercase letter *c* to its uppercase equivalent. All other characters are left unchanged.

RETURNS

toupper returns the corresponding uppercase letter, or the unchanged character.

EXAMPLE

To convert a string to uppercase:

```
for (; *s; ++s)
     *s = toupper(*s);
```

SEE ALSO

islower, **isupper**, **toupper**

NOTES

Make sure the argument *c* is within a restricted range, as described under the header **<ctype.h>**.

NAME

ims.h – header file for IMS access

SYNOPSIS

```
#include <ims.h>
```

FUNCTION

The header file **<ims.h>** declares the functions you use to communicate with IMS. You include **<ims.h>** only if you are writing a C program for an IMS environment. You must link a C program in a special way if it is to run in an IMS environment. See Chapter 4, "Linking and Loading Programs" in your *C Compiler User's Guide for MVS, MVS/XA* (SC09-1129) for information on linking programs to run under IMS.

<ims.h> defines the following macro:

_pcblist – A modifiable lvalue macro that designates the data object the C runtime environment will use to store a pointer to the list of PCB's (Program Communication Blocks) obtained from IMS. **_pcblist** is of type *pointer to array of void* *. It has static lifetime.

_pcblist points to the first element of an array of pointers to PCB's. The C runtime environment writes a NULL pointer after the last PCB pointer in the array. If there are no PCB pointers, then **(*_pcblist)[0] == NULL**. Therefore, **(*_pcblist)[0]** points to the first PCB, **(*_pcblist)[1]** points to the second PCB, and so forth. You may assign a value to **_pcblist** or access its stored value. You may not take its address. PCB's are described in *IMS/VS Version 1 Application Programming* (SH20-9026).

There are three different ways to invoke your C program under IMS. If your program is not running under IMS, then **(*_pcblist)[0] == NULL** at program startup. If your program is running under IMS and was invoked by IMS, then **_pcblist** is set at program startup as described above. If your program is running under IMS and was invoked from a C program using the **system** function, then **_pcblist** is set to point to a copy of the array pointed at by **_pcblist** in the program that called the **system** function.

<ims.h> declares the following functions:

asmtdli – Call IMS DL/I.

ctdli – Call IMS DL/I from C. **ctdli** calls the function **asmtdli**. It also returns a status code indicating success or failure.

asmtdli

NAME

asmtdli - call IMS DL/I

SYNOPSIS

```
#include <ims.h>
void asmtdli(int cmd, ...);
```

FUNCTION

asmtdli calls DL/I to perform the command *cmd* you specify. **asmtdli** uses the standard OS calling sequence as described in Chapter 8, "Interfaces with Other Languages," in your *C Compiler User's Guide for MVS, MVS/XA* (SC09-1129). The arguments to **asmtdli** are described in *IMS/VS Version 1 Application Programming* (SH20-9026).

asmtdli accepts a variable number of arguments. The first argument can optionally be a count of the number of arguments that follow.

If you want DL/I to modify an argument, write the argument as a pointer to a data object of the proper type. Otherwise, the compiler will create a temporary copy of the argument you specify and DL/I will modify that temporary copy.

RETURNS

Nothing. DL/I returns results by modifying the arguments to **asmtdli**.

SEE ALSO

ctdli

NAME

ctdli - call IMS DL/I

SYNOPSIS

```
#include <ims.h>
int ctdli(int cmd, ...);
```

FUNCTION

ctdli calls DL/I to perform the command *cmd* you specify. **ctdli** calls **asmtdli**, which uses the standard OS calling sequence as described in Chapter 8, "Interfaces with Other Languages," in your *C Compiler User's Guide for MVS, MVS/XA* (SC09-1129). The arguments to **ctdli** are the same as the arguments to **asmtdli**. The arguments to **asmtdli** are described in *IMS/VS Version 1 Application Programming* (SH20-9026).

ctdli accepts a variable number of arguments. The first argument can optionally be a count of the number of arguments that follow.

If you want DL/I to modify an argument, write the argument as a pointer to a data object of the proper type. Otherwise, the compiler will create a temporary copy of the argument you specify and DL/I will modify that temporary copy.

RETURNS

ctdli returns zero if the status field (bytes 11 and 12 of the PCB argument) is blank. Otherwise, the status field is taken as a two byte unsigned number widened to *int*, which is used as the return value from **ctdli**.

EXAMPLE

To illustrate a few of the possible DL/I calls:

```
typedef struct
    {
    char s_name[8];
    char s_qual[1];
    char s_keynam[8];
    char s_opr[2];
    char s_keyval[6];
    char s_endchr[1];
    } Qual_ssa;

typedef struct
    {
    char s_nameu[8];
    char s_blank[1];
    } Unq_ssa;

typedef struct
    {
    char ioa;
    /* ...*/
    } Ioa;
```

```
typedef struct
     {
     char dbname[8];
     char seglv[2];
     char statcd[2];
     char proco[4];
     int filler;
     char segnam[8];
     int lenkfb;
     int nsenssg;
     char keyfb;
     } Db_pcb;

static Qual_ssa qual_ssa = {
     {'R', 'O', 'O', 'T', ' ', ' ', ' ', ' '},
     {'('},
     {'K', 'E', 'Y', ' ', ' ', ' ', ' ', ' '},
     {' ', '='},
     {'v', 'v', 'v', 'v', 'v', 'v'},
     {')'}};

static Unq_ssa unq_ssa = {
     {'N', 'A', 'M', 'E', ' ', ' ', ' ', ' '},
     {' '}};

static Ioa mst_s_ioa;
static Ioa det_s_ioa;

/*  entry point from IMS (DL/I)
 */
void main(ac, av)
     int ac;
     char *av[];
     {
     Db_pcb *p_pcbm = (*_pcblist)[0];
     Db_pcb *p_pcbd = (unsigned long)(*_pcblist)[1]  \
               & 0x7FFFFFFF;

     if (p_pcbm)
          {
          ctdli(4, "GU  ", p_pcbd, &det_s_ioa, &qual_ssa);
          ctdli("GHU ", p_pcbm, &mst_s_ioa, &qual_ssa);
          ctdli(3, "GHN ", p_pcbm, &mst_s_ioa);
          ctdli("REPL", p_pcbm, &mst_s_ioa);
          }
     }
```

SEE ALSO

asmtdli, <ims.h>

NAME

limits.h – header file for environmental limits

SYNOPSIS

```
#include <limits.h>
```

FUNCTION

The header file **<limits.h>** defines various properties of the representations of the arithmetic types. You include **<limits.h>** if your program must generate code that depends upon the value of any of these limits. In the list of macros that follows, the actual values for System/370 are given in brackets following the macro name.

<limits.h> defines the following macros:

CHAR_BIT – [8] the number of bits in a byte.

CHAR_MAX – [255] the maximum value for a data object of type *char*.

CHAR_MIN – [0] the minimum value for a data object of type *char*.

DBL_DIG – [15] the maximum number of decimal digits of precision for a data object of type *double*.

DBL_MAX_EXP – [76] the maximum power of 10 that System/370 can represent in a data object of type *double*.

DBL_MIN_EXP – [-76] the minimum power of 10 that System/370 can represent in a data object of type *double*.

DBL_RADIX – [16] the radix of *double* exponent representation.

DBL_ROUNDS – [0] whether *double* arithmetic rounds (1) or chops (0).

FLT_DIG – [6] the maximum number of decimal digits of precision for a data object of type *float*.

FLT_MAX_EXP – [76] the maximum power of 10 that System/370 can represent in a data object of type *float*.

FLT_MIN_EXP – [-76] the minimum power of 10 that System/370 can represent in a data object of type *float*.

FLT_RADIX – [16] the radix of *float* exponent representation.

FLT_ROUNDS – [0] whether *float* arithmetic rounds (1) or chops (0).

INT_MAX – [2,147,483,647] the maximum value for a data object of type *int*.

INT_MIN – [-2,147,483,648] the minimum value for a data object of type *int*.

LDBL_DIG – [15] the maximum number of decimal digits of precision for a data object of type *long double*.

LDBL_MAX_EXP – [76] the maximum power of 10 that System/370 can represent in a data object of type *long*

double.

LDBL_MIN_EXP – [-76] the minimum power of 10 that System/370 can represent in a data object of type *long double.*

LDBL_RADIX – [16] the radix of *long double* exponent representation.

LDBL_ROUNDS – [0] whether *long double* arithmetic rounds (1) or chops (0).

LONG_MAX – [2,147,483,647] the maximum value for a data object of type *long.*

LONG_MIN – [-2,147,483,648] the minimum value for a data object of type *long.*

SCHAR_MAX – [127] the maximum value for a data object of type *signed char.*

SCHAR_MIN – [-128] the minimum value for a data object of type *signed char.*

SHRT_MAX – [32,767] the maximum value for a data object of type *short.*

SHRT_MIN – [-32,768] the minimum value for a data object of type *short.*

UCHAR_MAX – [255] the maximum value for a data object of type *unsigned char.*

UINT_MAX – [4,294,987,295] the maximum value for a data object of type *unsigned int.*

ULONG_MAX – [4,294,987,295] the maximum value for a data object of type *unsigned long.*

USHRT_MAX – [65,535] the maximum value for a data object of type *unsigned short.*

All of these macros expand to constant integer expressions. You can use all of them in **#if** preprocessor directives except **UINT_MAX** and **ULONG_MAX.**

NAME

math.h – header file for mathematical functions

SYNOPSIS

```
#include <math.h>
```

FUNCTION

The header file **<math.h>** declares a number of functions commonly used in mathematics.

Many of these functions report a "range error" if the value of the function cannot be well approximated as a value of type *double*. A range error can be caused by either floating underflow or floating overflow. Some also report a "domain error" if the value of an argument is outside the range for which the function is defined. In either case, **errno** is set to a nonzero value, and a special value is returned by the function. If the error is a range error caused by floating underflow, the special value returned is zero. The description of each function documents when these errors can occur and what special values are returned for errors other than floating underflow. The macro **errno** is defined in the header file **<stdefs.h>**.

<math.h> defines the following macros:

EDOM – the value stored in **errno** when a domain error occurs. It expands to a constant integer expression.

ERANGE – the value stored in **errno** when a range error occurs. It expands to a constant integer expression.

HUGE_VAL – the value returned in place of a value that is too large to represent as type **double**. **HUGE_VAL** expands to an rvalue of type *double*. It is not necessarily a constant expression that you can use in a data initializer. It will not compare equal to a value of type *float*.

signgam – an rvalue whose value is the sign of *gamma* x from the last call to the function **gamma**. Its type is *int*.

<math.h> declares the following functions:

abs – integer absolute value.

acos – arccosine.

asin – arcsine.

atan – arctangent.

atan2 – arctangent of ratio.

ceil – nearest more positive integer.

cos – cosine.

cosh – hyperbolic cosine.

erf – error function.

erfc – complementary error function.

exp – exponential.

fabs – floating absolute value.

floor – nearest more negative integer.

fmod – floating modulus.

frexp – extract fraction from exponent.

gamma – log gamma.

hypot – hypotenuse.

j0 – j0 Bessel.

j1 – j1 Bessel.

jn – jn Bessel.

ldexp – scale exponent.

log – natural logarithm.

log10 – common logarithm.

modf – extract fraction and integer.

pow – raise to a power.

sin – sine.

sinh – hyperbolic sine.

sqrt – square root.

tan – tangent.

tanh –hyperbolic tangent.

y0 – y0 Bessel.

y1 – y1 Bessel.

yn – yn Bessel.

SEE ALSO

<stdefs.h>

NAME

abs – integer absolute value

SYNOPSIS

```
#include <math.h>
int abs(int i);
```

FUNCTION

abs calculates the absolute value of *i*.

abs does not check to see that the result is representable. On System/370, the integer **0x80000000** is the largest negative integer value. If you negate it, it overflows to become the same large negative value. This is the only value that overflows when you negate it.

RETURNS

abs returns the absolute value of *i*.

EXAMPLE

To print out a debit or credit balance:

```
printf("balance %d%s\n", abs(bal),
     (bal < 0) ? "CR" : (bal == 0) ? "" : "DB");
```

SEE ALSO

fabs

NAME

acos – arccosine

SYNOPSIS

```
#include <math.h>
double acos(double x);
```

FUNCTION

acos computes the angle, in radians, whose cosine is *x*. If *x* is outside the range [−1, 1], a domain error occurs.

RETURNS

acos returns the nearest internal representation to *acos x*, in the range [0, π]. If a domain error occurs, it sets **errno** to **EDOM** and returns zero.

EXAMPLE

To compute the polar angle of the vector to (**x**, **y**):

```
theta = acos (x / hypot(x, y));
if (y < 0)
     theta = -theta;
```

SEE ALSO

asin, **atan**, **atan2**

NAME

asin - arcsine

SYNOPSIS

```
#include <math.h>
double asin(double x);
```

FUNCTION

asin computes the angle, in radians, whose sine is *x*. If *x* is outside the range [-1, 1], a domain error occurs.

RETURNS

asin returns the nearest internal representation to *asin x*, in the range [$-\pi/2$, $\pi/2$]. If a domain error occurs, it sets **errno** to **EDOM** and returns zero.

EXAMPLE

To compute the polar angle of the vector to (**x**, **y**):

```
theta = asin(y / hypot(x, y));
if (x < 0)
     theta = 3.1419265F - theta;
```

SEE ALSO

acos, **atan**, **atan2**

NAME

atan - arctangent

SYNOPSIS

```
#include <math.h>
double atan(double x);
```

FUNCTION

atan computes the angle, in radians, whose tangent is *x*.

RETURNS

atan returns the nearest internal representation to *atan x*, in the range [$-\pi/2$, $\pi/2$].

EXAMPLE

To convert to polar coordinates in the right half plane:

```
rho = hypot(x, y);
theta = atan(y / x);          /* also see atan2 */
```

SEE ALSO

acos, **asin**, **atan2**

NAME

atan2 – arctangent of ratio

SYNOPSIS

```
#include <math.h>
double atan2(double y, double x);
```

FUNCTION

atan2 computes the angle, in radians, of the vector that passes through the origin (0, 0) and the point (x, y). If both x and y are zero, a domain error occurs.

RETURNS

atan2 returns the nearest internal representation to *atan* y/x, in the range $[-\pi, \pi]$. If a domain error occurs, it sets **errno** to **EDOM** and returns zero.

EXAMPLE

To convert any Cartesian coordinates to polar coordinates:

```
rho = hypot(x, y);
theta = atan2(y, x);
```

SEE ALSO

acos, **asin**, **atan**, **hypot**

ceil

NAME

ceil – nearest more positive integer

SYNOPSIS

```
#include <math.h>
double ceil(double x);
```

FUNCTION

ceil computes the smallest integer greater than or equal to *x*.

RETURNS

ceil returns the smallest integer greater than or equal to *x*.

EXAMPLE

Some sample values are:

x	*ceil(x)*
5.1	6.0
5.0	5.0
4.9	5.0
0.0	0.0
−4.9	−4.0
−5.0	−5.0
−5.1	−5.0

SEE ALSO

floor

NAME

cos – cosine

SYNOPSIS

```
#include <math.h>
double cos(double x);
```

FUNCTION

cos computes the cosine of *x*, expressed in radians.

RETURNS

cos returns the nearest internal representation to *cos x*. As *x* increases in magnitude, its angular resolution becomes progressively less precise. For magnitudes of *x* that are large in comparison to π, you may find that the result **cos** returns is not useful, even though the result still closely approximates *cos x*.

EXAMPLE

To rotate a vector through the angle **theta**:

```
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

SEE ALSO

sin, **tan**

cosh

NAME

cosh – hyperbolic cosine

SYNOPSIS

```
#include <math.h>
double cosh(double x);
```

FUNCTION

cosh computes the hyperbolic cosine of *x*. A range error may occur.

RETURNS

cosh returns the nearest internal representation to *cosh x*. If a range error occurs, it sets **errno** to **ERANGE** and returns **HUGE_VAL** for values too large to represent.

EXAMPLE

To raise **cosh x + sinh x** to the **n**th power:

```
demoivre = cosh(n * x) + sinh(n * x);
```

SEE ALSO

exp, **sinh**, **tanh**

NAME

erf – error function

SYNOPSIS

```
#include <math.h>
double erf(double x);
```

FUNCTION

erf computes the error function of *x*.

RETURNS

erf returns the nearest internal representation to *erf x*.

EXAMPLE

To compute a probability interval:

```
if (1.0 < x)
     prob = erfc(x) - erfc(x + epsilon);
else
     prob = erf(x + epsilon) - erf(x);
```

SEE ALSO

erfc

NAME

erfc – complementary error function

SYNOPSIS

```
#include <math.h>
double erfc(double x);
```

FUNCTION

erfc computes the complementary error function of *x*, as defined by *(1 – erf x)*.

RETURNS

erfc returns the nearest internal representation to *erfc x*.

EXAMPLE

To compute a probability interval:

```
if (1.0 < x)
     prob = erfc(x) - erfc(x + epsilon);
else
     prob = erf(x + epsilon) - erf(x);
```

SEE ALSO

erf

NAME

exp - exponential

SYNOPSIS

```
#include <math.h>
double exp(double x);
```

FUNCTION

exp computes the exponential of *x*. A range error may occur.

RETURNS

exp returns the nearest internal representation to *exp x*. If a range error occurs, it sets **errno** to **ERANGE** and returns **HUGE_VAL** for values too large to represent.

EXAMPLE

To compute the hyperbolic sine of **x**:

```
y = (exp(x) - exp(-x)) / 2.0;      /* also see sinh */
```

SEE ALSO

log

fabs

NAME

fabs - floating absolute value

SYNOPSIS

```
#include <math.h>
double fabs(double x);
```

FUNCTION

fabs computes the absolute value of *x*.

RETURNS

fabs returns the absolute value of *x*.

EXAMPLE

Some values are:

x	*fabs(x)*
5.0	5.0
0.0	0.0
-3.7	3.7

SEE ALSO

abs

NAME

floor – nearest more negative integer

SYNOPSIS

```
#include <math.h>
double floor(double x);
```

FUNCTION

floor computes the largest integer less than or equal to *x*.

RETURNS

floor returns the largest integer less than or equal to *x*.

EXAMPLE

Some sample values are:

x	*floor(x)*
5.1	5.0
5.0	5.0
4.9	4.0
0.0	0.0
−4.9	−5.0
−5.0	−5.0
−5.1	−6.0

SEE ALSO

ceil

fmod

NAME

fmod – floating modulus

SYNOPSIS

```
#include <math.h>
double fmod(double x, double y);
```

FUNCTION

fmod computes the "modulus" of *x* and *y*. The modulus is defined as:

*fmod(x, y) == x – (i * y)*

where *i* is an integer value and

fabs(fmod(x, y)) < fabs(y)

The value of **fmod** is defined even if *i* is not representable as a value of type *double*.

RETURNS

fmod returns the modulus of *x* and *y*. The sign of the return value has the same sign as *x*. If *y* is zero, **fmod** returns *x*.

EXAMPLE

Some values are:

x	*y*	*fmod(x, y)*
5.5	5.0	0.5
5.0	5.0	0.0
–1.3	0.0	–1.3
–5.5	5.0	–0.5

NAME

frexp - extract fraction from exponent

SYNOPSIS

```
#include <math.h>
double frexp(double val, int *exp);
```

FUNCTION

frexp partitions *val* into a fraction whose magnitude lies in the half open interval [1/2, 1), multiplied by two raised to an integer power. If *val* is zero, the fraction and exponent are both zero.

RETURNS

frexp stores the integer power at **exp*, and returns the fraction as the value of the function.

EXAMPLE

To implement a *float* square root function:

```
float fsqrt(x)
     float x;
     {
     float y;
     int i, n;

     x = frexp(x, &n);
     if (x)
          {
          y = x;
          for (i = 4; 0 <= --i; )
               y = (x / y + y) * 0.5F;
                    /* divide and average */
          x = y;
          }
     if (n & 1)
          x *= 1.4142136F;
     return (ldexp(x, n / 2));
     }
```

SEE ALSO

ldexp, modf

gamma

NAME

gamma – log gamma

SYNOPSIS

```
#include <math.h>
double gamma(double x);
```

FUNCTION

gamma computes the natural log of the magnitude of the gamma function. If *x* is an integer less than or equal to zero, a domain error occurs. A range error may occur.

RETURNS

If *gamma x* is positive, **gamma** stores the value 1 in an *int* data object whose value can be accessed by the rvalue macro **signgam**. It stores the value −1 if *gamma x* is negative. It then returns the nearest internal representation to *log* |*gamma x*|. If a domain error occurs, it sets **errno** to **EDOM** and returns **HUGE_VAL**. If a range error occurs, it sets **errno** to **ERANGE** and returns **HUGE_VAL** for values too large to represent.

EXAMPLE

To compute the function *gamma x*:

```
y = exp(gamma(x));
if (signgam < 0)
     y = -y;
```

NAME

hypot - hypotenuse

SYNOPSIS

```
#include <math.h>
double hypot(double x, double y);
```

FUNCTION

hypot computes the hypotenuse of a right triangle whose sides are *x* and *y*. A range error may occur.

RETURNS

hypot returns the nearest internal representation to the hypotenuse of *x* and *y*. If a range error occurs, it sets **errno** to **ERANGE** and returns **HUGE_VAL** for values too large to represent.

EXAMPLE

To convert any Cartesian coordinates to polar coordinates:

```
rho = hypot(x, y);
theta = atan2(y, x);
```

SEE ALSO

atan2

j0

NAME

j0 - j0 Bessel

SYNOPSIS

```
#include <math.h>
double j0(double x);
```

FUNCTION

j0 computes the Bessel function of *x* of the first kind, of order 0. A range error may occur, but only as the result of a floating underflow.

RETURNS

j0 returns the nearest internal representation to *j0 x*. If a range error occurs, it sets **errno** to **ERANGE** and returns zero.

EXAMPLE

To compute *j2 x*:

```
j2x = (2.0 / x) * j1(x) - j0(x);  /* also see jn */
```

SEE ALSO

j1, **jn**, **y0**, **y1**, **yn**

NAME

j1 - j1 Bessel

SYNOPSIS

```
#include <math.h>
double j1(double x);
```

FUNCTION

j1 computes the Bessel function of *x* of the first kind, of order 1. A range error may occur, but only as the result of a floating underflow.

RETURNS

j1 returns the nearest internal representation to *j1 x*. If a range error occurs, it sets **errno** to **ERANGE** and returns zero.

EXAMPLE

To compute *j2 x*:

```
j2x = (2.0 / x) * j1(x) - j0(x);  /* also see jn */
```

SEE ALSO

j0, **jn**, **y0**, **y1**, **yn**

NAME

jn – jn Bessel

SYNOPSIS

```
#include <math.h>
double jn(int n, double x);
```

FUNCTION

jn computes the Bessel function of x of the first kind, of order n. A range error may occur, but only as the result of a floating underflow.

RETURNS

jn returns the nearest internal representation to *jn x*. If a range error occurs, it sets **errno** to **ERANGE** and returns zero.

EXAMPLE

To compute *j2 x*:

```
j2x = jn(2, x);
```

SEE ALSO

j0, **j1**, **y0**, **y1**, **yn**

NAME

ldexp - scale exponent

SYNOPSIS

```
#include <math.h>
double ldexp(double x, int exp);
```

FUNCTION

ldexp multiplies *x* by two raised to the power *exp*. A range error may occur.

RETURNS

ldexp returns the computed product. If a range error occurs, it sets **errno** to **ERANGE** and returns **HUGE_VAL** for values positive and too large to represent. If the value is negative and too large, it returns **-HUGE_VAL**.

EXAMPLE

To implement a *float* square root function:

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

main()
     {
     float x, y;
     float fsqrt();

     x = 3.0;
     y = fsqrt(x);
     printf("the sqrt of %f is %f\n", x, y);
     }

float fsqrt(x)
     float x;
     {
     float y;
     int i, n;

     x = frexp(x, &n);
     if (x)
          {
          y = x;
          for (i = 4; 0 <= --i; )
               y = (x / y + y) * 0.5F;
               /* divide and average */
          x = y;
          }
     if (n & 1)
          x *= 1.4142136F;
     return (ldexp(x, n / 2));
     }
```

SEE ALSO

frexp, modf

NAME

log - natural logarithm

SYNOPSIS

```
#include <math.h>
double log(double x);
```

FUNCTION

log computes the natural logarithm of *x*. If *x* is less than zero, a domain error occurs. If *x* is zero, a range error occurs.

RETURNS

log returns the nearest internal representation to *log x*. If a domain error occurs, it sets **errno** to **EDOM** and returns zero. If a range error occurs, it sets **errno** to **ERANGE** and returns **-HUGE_VAL**.

EXAMPLE

To compute the hyperbolic arccosine of **x**:

```
arccosh = log(x + sqrt(x * x + 1));
```

SEE ALSO

exp, **log10**

log10

NAME

log10 - common logarithm

SYNOPSIS

```
#include <math.h>
double log10(double x);
```

FUNCTION

log10 computes the common logarithm of *x*. If *x* is less than zero, a domain error occurs. If *x* is zero, a range error occurs.

RETURNS

log10 returns the nearest internal representation to *log10 x*. If a domain error occurs, it sets **errno** to **EDOM** and returns zero. If a range error occurs, it sets **errno** to **ERANGE** and returns **-HUGE_VAL**.

EXAMPLE

To determine the number of digits needed to represent **x** if **x** is printed as a text string:

```
ndig = (int)log10(x)+1;
if (ndig < 1)
     ndig = 1;
```

SEE ALSO

log

NAME

modf – extract fraction and integer

SYNOPSIS

```
#include <math.h>
double modf(double val, double *pd);
```

FUNCTION

modf partitions *val* into an integer and a fraction whose magnitude is less than 1. The integer and fraction both have the same sign as *val*.

RETURNS

modf stores the integer part in the data object at ***pd**. It returns the fraction as the value of the function.

EXAMPLE

Some sample values are:

val	**pd*	*modf(val, pd)*
5.1	5	0.1
5.0	5	0.0
4.9	4	0.9
0.0	0	0.0
−1.4	−1	−0.4

SEE ALSO

frexp, **ldexp**

NAME

pow – raise to a power

SYNOPSIS

```
#include <math.h>
double pow(double x, double y);
```

FUNCTION

pow computes the value of *x* raised to the power *y*. If *x* is zero and *y* is less than or equal to zero, or if *x* is negative and *y* is not an integer, a domain error occurs. A range error may occur.

RETURNS

pow returns nearest internal representation to *x* raised to the power *y*. If a domain error occurs, it sets **errno** to **EDOM** and returns zero. If a range error occurs, it sets **errno** to **ERANGE** and returns **HUGE_VAL** for values positive and too large to represent. For values negative and too large to represent, **pow** returns **-HUGE_VAL**.

EXAMPLE

Some sample values are:

x	*y*	*pow(x, y)*
2.0	2.0	4.0
2.0	1.0	2.0
2.0	0.0	1.0
1.0	any	1.0
0.0	-2.0	domain error
-1.0	3.0	-1.0
-1.0	2.1	domain error

SEE ALSO

exp, **log**

NAME

sin - sine

SYNOPSIS

```
#include <math.h>
double sin(double x);
```

FUNCTION

sin computes the sine of *x*, expressed in radians.

RETURNS

sin returns the nearest internal representation to *sin x*. As *x* increases in magnitude, its angular resolution becomes progressively less precise. For magnitudes of *x* that are large in comparison to π, you may find that the result **sin** returns is not useful, even though the result still closely approximates *sin x*.

EXAMPLE

To rotate a vector through the angle **theta**:

```
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

SEE ALSO

cos, **tan**

sinh

NAME

sinh – hyperbolic sine

SYNOPSIS

```
#include <math.h>
double sinh(double x);
```

FUNCTION

sinh computes the hyperbolic sine of *x*. A range error may occur.

RETURNS

sinh returns the nearest internal representation to *sinh x*. If a range error occurs, it sets **errno** to **ERANGE** and returns **HUGE_VAL** for values too large to represent.

EXAMPLE

To raise **cosh x + sinh x** to the nth power:

```
demoivre = cosh(n * x) + sinh(n * x);
```

SEE ALSO

exp, **cosh**, **tanh**

NAME

sqrt - square root

SYNOPSIS

```
#include <math.h>
double sqrt(double x);
```

FUNCTION

sqrt computes the square root of *x*. If *x* is less than zero, a domain error occurs.

RETURNS

sqrt returns the nearest internal representation to *sqrt* x. If a domain error occurs, it sets **errno** to **EDOM** and returns zero.

EXAMPLE

To check whether **n** is a prime number (where **n** is greater than 2):

```
if (!(n & 01))
     return (1);
sq = sqrt((double)n);
for (div = 3; div < sq; div += 2)
     if (!(n % div))
          return (1);
return (0);
```

tan

NAME

tan – tangent

SYNOPSIS

```
#include <math.h>
double tan(double x);
```

FUNCTION

tan computes the tangent of *x*, expressed in radians. If the magnitude of *x* is too large to contain a fractional quadrant part, the return value is 0. A range error may occur.

RETURNS

tan returns the nearest internal representation to *tan x*. A large argument may return a meaningless value. If a range error occurs, it sets **errno** to **ERANGE** and returns **HUGE_VAL** for values too large to represent.

EXAMPLE

To compute the coordinate **y**, given **x** and the polar angle **theta**:

```
y = x * tan(theta);
```

SEE ALSO

cos, **sin**

NAME

tanh - hyperbolic tangent

SYNOPSIS

```
#include <math.h>
double tanh(double x);
```

FUNCTION

tanh computes the hyperbolic tangent of *x*.

RETURNS

tanh returns the nearest internal representation to *tanh x*.

EXAMPLE

To compute the hyperbolic tangent of **x**:

```
y = tanh(x);
```

SEE ALSO

exp, **cosh**, **sinh**

y0

NAME

y0 - y0 Bessel

SYNOPSIS

```
#include <math.h>
double y0(double x);
```

FUNCTION

y0 computes the Bessel function of *x* of the second kind, of order 0. If *x* is less than or equal to zero, a domain error occurs. A range error may occur, but only as the result of a floating underflow.

RETURNS

y0 returns the nearest internal representation to *y0 x*. If a domain error occurs, it sets **errno** to **EDOM** and returns **-HUGE_VAL**. If a range error occurs, it sets **errno** to **ERANGE** and returns zero.

EXAMPLE

To compute *y2 x*:

```
y2x = (2.0 / x) * y1(x) - y0(x);  /* also see yn */
```

SEE ALSO

j0, **j1**, **jn**, **y1**, **yn**

NAME

y1 - y1 Bessel

SYNOPSIS

```
#include <math.h>
double y1(double x);
```

FUNCTION

y1 computes the Bessel function of *x* of the second kind, of order 1. If *x* is less than or equal to zero, a domain error occurs. A range error may occur, but only as the result of a floating underflow.

RETURNS

y1 returns the nearest internal representation to *y1 x*. If a domain error occurs, it sets **errno** to **EDOM** and returns **-HUGE_VAL**. If a range error occurs, it sets **errno** to **ERANGE** and returns zero.

EXAMPLE

To compute *y2 x*:

```
y2x = (2.0 / x) * y1(x) - y0(x);  /* also see yn */
```

SEE ALSO

j0, **j1**, **jn**, **y0**, **yn**

NAME

yn - yn Bessel

SYNOPSIS

```
#include <math.h>
double yn(int n, double x);
```

FUNCTION

yn computes the Bessel function of *x* of the second kind, of order *n*. If *x* is less than or equal to zero, a domain error occurs. A range error may occur, but only as the result of a floating underflow.

RETURNS

yn returns the nearest internal representation to *yn x*. If a domain error occurs, it sets **errno** to **EDOM** and returns **-HUGE_VAL**. If a range error occurs, it sets **errno** to **ERANGE** and returns zero.

EXAMPLE

To compute *y2 x*:

```
y2x = yn(2, x);
```

SEE ALSO

j0, **j1**, **jn**, **y0**, **y1**

NAME

setjmp.h – header file for nonlocal jumps

SYNOPSIS

```
#include <setjmp.h>
```

FUNCTION

The header file **<setjmp.h>** declares functions which save and restore a calling environment. You use them to bypass the normal nesting of function calls and returns.

<setjmp.h> declares the following type:

jmp_buf – an array type that holds the information needed to save a calling environment.

<setjmp.h> declares the following functions:

longjmp – restore calling environment.

setjmp – save calling environment.

longjmp

NAME

longjmp - restore calling environment

SYNOPSIS

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

FUNCTION

longjmp restores the calling environment saved in *env* by **setjmp**. You must save a calling environment in *env* by calling **setjmp**, and your program must not have returned from the function containing the call to **setjmp** before calling **longjmp**.

When you call **longjmp**, it restores the values stored in *some* objects of storage class **register** or **auto** in the function containing the call to **setjmp**, to the values stored in them at the time of the **setjmp** call. There is no reliable way to determine which of these objects are restored, so you should not depend upon the stored values of register or automatic data objects that may have changed between the **setjmp** and **longjmp** calls. All other data objects are unchanged when the calling environment is restored.

longjmp bypasses the function call and return mechanism that other functions use. You use it to return early from an arbitrary nest of function calls. You can also use it within a signal handler.

RETURNS

longjmp never returns. Instead, program execution continues by returning again from the call to **setjmp** that saved the calling environment. If *val* has the value 0, **setjmp** returns 1. Otherwise, **setjmp** returns *val*.

EXAMPLE

You can write a generic signal handler as:

```
void handle(sig)
     int sig;
     {
     extern jmp_buf env;

     longjmp(env, sig); /* return from setjmp */
     }
```

SEE ALSO

setjmp

NOTES

When using the **setjmp/longjmp** function in combination with **signal(SIGINT)**, a permanent restriction causes control to return to the instruction that the program was executing when the signal occurred instead of to the **setjmp** call when **longjmp** is called.

NAME

setjmp - save calling environment

SYNOPSIS

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

FUNCTION

setjmp saves the calling environment in *env* for later use by the function **longjmp**. Do not write a call to **setjmp** as a subexpression. Write it as an *expression* statement, or as the expression in a *switch* or an *if* statement.

RETURNS

setjmp returns zero when you first call it. It saves the current calling environment in *env*. If a call to **longjmp** designates the same saved calling environment, **setjmp** returns again with the calling environment restored. The value returned is determined by the call to **longjmp**, as described for that function.

EXAMPLE

To handle signals from a generic signal handler:

```
switch (setjmp(env))
     {
case 0:
     break;          /* go do normal processing */
case SIGFPE:
     printf("floating point exception\n");
     exit(0);
case SIGILL:
case SIGSEGV:
     printf("bad news\n");
     exit(1);
case SIGINT:
     printf("interrupted\n");
     exit(0);
default:
     printf("unknown signal\n");
     exit(1);
     }
```

SEE ALSO

longjmp

signal.h

NAME

signal.h – header file for exceptional condition handling

SYNOPSIS

```
#include <signal.h>
```

FUNCTION

The header file **<signal.h>** declares functions for reporting and handling exceptional conditions, or "signals." The runtime environment converts certain hardware exceptions to C signals. You can generate an asynchronous signal from your keyboard. In your C program you can report any signal by calling the function **kill**. You include **<signal.h>** to alter the default reporting and handling of signals.

<signal.h> defines the following macros:

SIG_DFL – the function pointer value you use on a call to **signal** to request default signal handling. This expands to a constant expression of type *pointer to function returning void.*

SIG_ERR – the function pointer value returned by **signal** to report an error. This expands to a constant expression of type *pointer to function returning void.*

SIG_IGN – the function pointer value you use on a call to **signal** to request that a signal be ignored. This expands to a constant expression of type *pointer to function returning void.*

SIGABRT – the signal value that forces your program to terminate abnormally.

SIGFPE – the signal value that reports a floating point exception.

SIGILL – the signal value that reports an invalid instruction.

SIGINT – the signal value that reports an asynchronous keyboard interrupt.

SIGSEGV – the signal value that reports a memory protection violation.

SIGSTACK – the signal value that reports on function call stack overflow.

SIGTERM – the signal value that forces your program to terminate quietly.

SIGUSR1 – a signal value reserved for your use.

SIGUSR2 – a signal value reserved for your use.

<signal.h> declares the following functions:

kill – report a signal.

signal – specify handling of a signal.

All macros that define signal values expand to constant integer expressions.

NAME

kill – send a signal

SYNOPSIS

```
#include <signal.h>
int kill(int pid, int sig);
```

FUNCTION

kill sends the signal *sig*. *pid* must be zero. The signals that **kill** sends are:

Name	*Meaning*
SIGABRT	force abnormal termination
SIGFPE	floating point exception
SIGILL	invalid instruction
SIGINT	keyboard interrupt
SIGSEGV	memory protection violation
SIGSTACK	stack overflow
SIGTERM	force quiet termination
SIGUSR1	reserved for your use
SIGUSR2	reserved for your use

Macros for all of these signal values are defined in the header file **<signal.h>**.

The C runtime environment uses the SPIE system service to catch exceptions. These are appropriately mapped to calls to **kill** that send the signals **SIGFPE**, **SIGILL**, and **SIGSEGV**:

Program interruption code	*Hardware exception name*	*C signal code*
1	Operation exception	SIGILL
2	Privileged operation exception	SIGILL
3	Execute exception	SIGILL
4	Protection exception	SIGSEGV
5	Addressing exception	SIGSEGV
6	Specification exception	SIGSEGV
7	Data exception	SIGFPE
8	Fixed point overflow exception	(masked)
9	Fixed point divide exception	SIGFPE
10	Decimal overflow exception	SIGFPE
11	Decimal divide exception	SIGFPE
12	Exponent overflow exception	SIGFPE
13	Exponent underflow exception	(masked)
14	Significance exception	(masked)
15	Floating point divide exception	SIGFPE

The C library uses the STAX system service to capture asynchronous keyboard interrupts. These call **kill** to send the signal **SIGINT**.

RETURNS

kill returns zero if *sig* has a proper value and if the handler for the signal returns. If *sig* does not have a proper value, **kill** returns a nonzero value.

EXAMPLE

To abort processing:

```
kill(0, SIGABRT);
```

SEE ALSO

abort, **signal**

NAME

signal - specify handling of a signal

SYNOPSIS

```
#include <signal.h>
void (*signal (int sig, void (*func)()))();
```

FUNCTION

signal changes the handling of the signal *sig* according to the value of *func*. The description of **kill** lists the valid values for *sig*.

If *func* has the value **SIG_DFL**, "default" handling takes effect for the specified signal. Default handling is to write an error message to the standard error stream and terminate your program abnormally. At program startup, default handling takes effect for all signals.

If *func* has the value **SIG_IGN**, the signal is subsequently ignored.

Otherwise *func* must be a pointer to a function that handles future reports of the specified signal. When **kill** reports a signal, the handler function is called with the expression:

(*func)(*sig*)

where *sig* is the signal reported by **kill**.

Your handler function should do as little as possible. It has only limited storage for calling other functions. The best signal handler simply alters the value stored in a scalar volatile static data object and returns. It may also call **longjmp** to restore an earlier calling environment. If a signal handler interrupts an input/output operation, as in response to an asynchronous keyboard signal, the stream or its associated file may be left in an inconsistent state.

RETURNS

signal returns the previous handler function *func* if successful. Otherwise it returns the value **SIG_ERR**.

EXAMPLE

To turn off keyboard interrupts:

```
signal(SIGINT, SIG_IGN);
```

SEE ALSO

abort, **kill**

stdarg.h

NAME

stdarg.h – header file for walking argument lists

SYNOPSIS

```
#include <stdarg.h>
```

FUNCTION

The header file **<stdarg.h>** defines macros that you use to "walk" an argument list to a function that accepts a variable number of arguments.

<stdarg.h> declares the following type:

va_list – an array type that holds information needed by the macros **va_arg** and **va_end**. A data object of this type must be first initialized by the macro **va_start**.

<stdarg.h> defines the following macros:

va_arg – get pointer to next argument in list.

va_end – terminate walking argument list.

va_start – initiate walking argument list.

NAME

va_arg – get pointer to next argument in list

SYNOPSIS

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

FUNCTION

The macro **va_arg** is an rvalue that computes the value of the next argument in a variable length argument list. Information on the argument list is stored in the array data object *ap*. You must first initialize *ap* with the macro **va_start**, and compute all earlier arguments in the list by expanding **va_arg** for each argument.

The type of the next argument is given by the type name *type*. The type name must be the same as the type of the next argument. Remember that the compiler widens an arithmetic argument to *int*, and converts an argument of type *float* to *double*. You write the type after conversion. Write **int** instead of **char** and **double** instead of **float**.

Do not write a type name that contains any parentheses. Use a type definition, if necessary, as in

```
typedef int (*Pfi)();
     /* pointer to function returning int */
.....
fun_ptr = va_arg(ap, Pfi);
     /* get function pointer argument */
```

RETURNS

va_arg expands to an rvalue of type *type*. Its value is the value of the next argument. It alters the information stored in *ap* so that the next expansion of **va_arg** accesses the argument following.

EXAMPLE

To write multiple strings to a file:

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

main()
     {
     void strput();

     strput(stdout, "This is one string\n",      \
          "and this is another...\n", 0);
     }

void strput(FILE *pf, ...);
```

```
void strput(pf)
     FILE *pf;
     {
     char *ptr;
     va_list va;

     va_start(va, pf);
     while (ptr = va_arg(va, char *))
          fputs(ptr, pf);
     va_end(va);
     }
```

SEE ALSO

va_end, va_start

NAME

va_end - terminate walking argument list

SYNOPSIS

```
#include <stdarg.h>
va_end(va_list ap);
```

FUNCTION

va_end is a macro which you must expand if you expand the macro **va_start** within a function that contains a variable length argument list. Information on the argument list is stored in the data object designated by *ap*. Designate the same data object in both **va_start** and **va_end**.

You expand **va_end** after you have accessed all argument values with the macro **va_arg**, before your program returns from the function that contains the variable length argument list. After you expand **va_end**, do not expand **va_arg** with the same *ap*. You need not expand **va_arg** within the function that contains the variable length argument list.

You must write an expansion of **va_end** as an *expression* statement containing a function call. The call must be followed by a semicolon.

RETURNS

Nothing. **va_end** expands to a statement, not an expression.

EXAMPLE

To write multiple strings to a file:

```
void strput(FILE *pf, ...);
void strput(pf)
     FILE *pf;
     {
     char *ptr;
     va_list va;

     va_start(va, pf);
     while (ptr = va_arg(va, char *))
          fputs(ptr, pf);
     va_end(va);
     }
```

SEE ALSO

va_arg, va_start

va_start

NAME

va_start - initiate walking argument list

SYNOPSIS

```
#include <stdarg.h>
va_start(va_list ap, parmN);
```

FUNCTION

va_start is a macro which you must expand before you expand the macro **va_arg**. It initializes the information stored in the data object designated by *ap*. The argument *parmN* must be the identifier you declare as the name of the last specified argument in the variable length argument list for the function. In the function prototype for the function, *parmN* is the argument name you write just before the **, ...**

The type of *parmN* must be one of the types assumed by an argument passed in the absence of a prototype. Its type must not be *float* or *char*. Also, *parmN* cannot have storage class **register**.

If you expand **va_start**, you must expand the macro **va_end** before your program returns from the function containing the variable length argument list.

You must write an expansion of **va_start** as an *expression* statement containing a function call. The call must be followed by a semicolon.

RETURNS

Nothing. **va_start** expands to a statement, not an expression.

EXAMPLE

To write multiple strings to a file:

```
void strput(FILE *pf, ...);
void strput(pf)
     FILE *pf;
     {
     char *ptr;
     va_list va;

     va_start(va, pf);
     while (ptr = va_arg(va, char *))
          fputs(ptr, pf);
     va_end(va);
     }
```

SEE ALSO

va_arg, va_end

NAME

stdefs.h – header file for common definitions

SYNOPSIS

```
#include <stdefs.h>
```

FUNCTION

The header file **<stdefs.h>** declares types and defines macros that are widely used.

<stdefs.h> defines the following types:

ptrdiff_t – the signed integer type the compiler chooses for the difference between two pointers.

size_t – the unsigned integer type the compiler chooses for the result of the **sizeof** operator.

<stdefs.h> defines the following macros:

NULL – a constant pointer expression with value 0. You can use **NULL** as a data object pointer argument to a function whether or not it checks the types of its arguments.

errno – a modifiable lvalue that designates the data object used for storing error codes. Its type is *int*, it has static lifetime, and its initial stored value is 0. You may assign values to **errno** or access its stored value. You may *not* take its address. Any C library function may report an error by storing a nonzero value in **errno**. No C library function will store the value 0 in **errno**.

stdio.h

NAME

stdio.h – header file for stream I/O

SYNOPSIS

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

FUNCTION

The header file **<stdio.h>** declares functions used to perform input and output.

<stdio.h> defines the following type:

FILE – a structure type which holds information for controlling a stream connected to an open file. The address of a data object of type **FILE** is significant. You cannot assign the value of a data object of type **FILE** to another data object and use the address of that data object in place of the address of the original data object.

<stdio.h> defines the following macros:

_IOFBF – the value you use as the third argument to **setvbuf** to specify full buffering. It expands to a constant integer expression.

_IOLBF – the value you use as the third argument to **setvbuf** to specify line buffering. It expands to a constant integer expression.

_IONBF – the value you use as the third argument to **setvbuf** to specify no buffering. It expands to a constant integer expression.

BUFSIZ – the default size in bytes of a stream buffer. It expands to a constant integer expression.

EOF – the value returned by a function to signal that end of file is encountered on a stream. It expands to a constant integer expression.

L_tmpnam – the size in bytes of an array big enough to hold a temporary file name returned by **tmpnam**. It expands to a constant integer expression.

SEEK_CUR – the value you use as the third argument to **fseek** to specify positioning relative to the current file pointer. It expands to a constant integer expression.

SEEK_END – the value you use as the third argument to **fseek** to specify positioning relative to the end of the file. It expands to a constant integer expression.

SEEK_SET – the value you use as the third argument to **fseek** to specify positioning relative to the beginning of the file. It expands to a constant integer expression.

SYS_OPEN – the maximum number of files that can be open at the same time. It expands to a constant integer expression.

TMP_MAX – the minimum number of unique file names that **tmpnam** generates. It expands to a constant integer expression. Its value is at least 25.

stderr – an rvalue whose value is the address of the data object used for controlling the standard error stream. Its type is *pointer to* **FILE**. It is not a constant expression. You may not use it in a data initializer.

stdin – an rvalue whose value is the address of the data object used for controlling the standard input stream. Its type is *pointer to* **FILE**. It is not a constant expression. You may not use it in a data initializer.

stdout – an rvalue whose value is the address of the data object used for controlling the standard output stream. Its type is *pointer to* **FILE**. It is not a constant expression. You may not use it in a data initializer.

<stdio.h> declares the following functions:

clearerr – reset end of file and error indicators.

fclose – close a file.

feof – test end of file indicator.

ferror – test read/write error indicator.

fflush – flush output stream.

fgetc – read a character from input stream.

fgets – read a text line from input stream.

fopen – open a file.

fprintf – write formatted arguments to output stream.

fputc – write a character to output stream.

fputs – write a text line to output stream.

fread – read records from input stream.

freopen – open a file with existing stream.

fscanf – read formatted arguments from input stream.

fseek – set file pointer.

ftell – get file pointer.

fwrite – write records to output stream.

getc – read a character from input stream.

getchar – read a character from standard input.

gets – read a text line from standard input.

perror – map error number.

printf – write formatted arguments to standard output.

putc – write a character to output stream.

putchar – write a character to standard output.

puts – write a text line to standard output.

remove – remove a file.

rename – change file name.

rewind – set file pointer to beginning of file.

scanf – read formatted arguments from standard input.

setbuf – set stream buffer.

setvbuf – set stream buffering strategy.

sprintf – write formatted arguments to a string.

sscanf – read formatted arguments from a string.

tmpfile – create temporary file.

tmpnam – generate temporary file name.

ungetc – push character back into input stream.

vfprintf – write formatted argument list to output stream.

vprintf – write formatted argument list to standard output.

vsprintf – write formatted argument list to a string.

The functions **getc** and **putc** may each be masked by a macro definition that behaves differently from a call on the underlying function. The macro version of each expands to an expression that evaluates its *pointer to* **FILE** argument more than once. That argument must not have side effects. The equivalent functions **fgetc** and **fputc**, on the other hand, are *never* masked by macro definitions. You may use these functions instead of **getc** and **putc**, or undefine the identifiers **getc** and **putc** with a **#undef** preprocessor directive, to avoid the problem of argument side effects.

If you call any of these functions that expect an argument of type *pointer to* **FILE**, the argument must be **stderr**, **stdin**, **stdout**, or a value returned by an earlier call to **fopen**. Otherwise, the C library sets **errno** to a nonzero value.

SEE ALSO

<stdefs.h>

NAME

clearerr – reset end of file and error indicators

SYNOPSIS

```
#include <stdio.h>
void clearerr(FILE *pf);
```

FUNCTION

clearerr resets the end of file and read/write error indicators for the stream controlled by the FILE structure pointed at by *pf*. **clearerr**, **fopen**, **freopen**, and **rewind** are the only C library functions that clear these indicators.

RETURNS

Nothing.

EXAMPLE

To check and clear an error condition:

```
if (ferror(fp))
    {
    printf("bad record, skipped\n");
    clearerr(fp);
    }
```

SEE ALSO

feof, **ferror**, **rewind**

fclose

NAME

fclose – close a file

SYNOPSIS

```
#include <stdio.h>
int fclose(FILE *pf);
```

FUNCTION

fclose closes the file under control of the FILE structure pointed at by *pf*. If your program opens a file for writing or updating, **fclose** flushes any remaining output before closing the file. If the C library allocated the associated buffer dynamically, **fclose** frees it. **fclose** then invalidates the FILE structure at *pf* until the C library uses it to control another open file.

RETURNS

fclose returns zero if it can close the file successfully.

EXAMPLE

To finish up if no more input is forthcoming:

```
if (feof(fp))
    {
    fclose(fp);
    exit(errcount != 0);
    }
```

SEE ALSO

fopen, **freopen**

NAME

feof - test end of file indicator

SYNOPSIS

```
#include <stdio.h>
int feof(FILE *pf);
```

FUNCTION

feof tests the end of file indicator for the file controlled by the FILE structure pointed at by *pf*.

RETURNS

feof returns a nonzero value if the end of file indicator is set, or if *pf* does not point at a FILE structure controlling an open file.

EXAMPLE

To check a token stream for proper termination:

```
if (tok == FIN && !feof(fp))
    printf("missing EOF in data file\n");
```

SEE ALSO

clearerr, ferror

ferror

NAME

ferror – test read/write error indicator

SYNOPSIS

```
#include <stdio.h>
int ferror(FILE *pf);
```

FUNCTION

ferror tests the read/write error indicator for the file controlled by the FILE structure pointed at by *pf*.

RETURNS

ferror returns a nonzero value if the read/write error indicator is set, or if *pf* does not point at a FILE structure controlling an open file.

EXAMPLE

To write an array and check for errors:

```
fwrite(a, sizeof a[0], n, pf);
if (ferror(pf))
    printf("error writing file\n");
```

SEE ALSO

clearerr, feof

NAME

fflush - flush output stream

SYNOPSIS

```
#include <stdio.h>
int fflush(FILE *pf);
```

FUNCTION

fflush drains any unwritten data in the output buffer for the stream controlled by the FILE structure pointed at by *pf*. If the output buffer is empty, or the current buffer is opened only for reading, **fflush** does nothing.

RETURNS

fflush returns a nonzero value if a write error occurs, or if *pf* does not point at a FILE structure controlling an open file.

EXAMPLE

To prompt on an interactive device:

```
fprintf(pfout, "are you sure? ");
fflush(pfout);
fgets(answer_buf, sizeof answer_buf, pfin);
```

SEE ALSO

fclose, **fopen**, **freopen**, **setvbuf**

fgetc

NAME

fgetc - read a character from input stream

SYNOPSIS

```
#include <stdio.h>
int fgetc(FILE *pf);
```

FUNCTION

fgetc obtains the next input character, if any, from the stream controlled by the FILE structure pointed at by *pf*. **fgetc** then increments the associated file pointer by one byte.

RETURNS

fgetc returns the value of the next character from the input stream pointed at by *pf*. The character value is type cast to *unsigned char*. If **fgetc** encounters end of file or a read error, or if *pf* does not point at a FILE structure controlling an open file, it returns the value **EOF**.

EXAMPLE

To copy a text file, character by character:

```
FILE *ipf, *opf;
int c;

ipf = fopen("infile.data", "r");
opf = fopen("outfile.data", "w");
while ((c = fgetc(ipf)) != EOF)
     fputc(c, opf);
```

SEE ALSO

fputc, **getc**, **getchar**, **putc**, **putchar**

NOTES

<stdio.h> does not define **fgetc** as a macro. **getc** is superior to **fgetc** in this regard, even though you must exercise care in calling **getc**.

NAME

fgets - read a text line from input stream

SYNOPSIS

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *pf);
```

FUNCTION

fgets copies characters from the file controlled by the FILE structure pointed at by *pf* to the *n* character buffer starting at *s*. It copies characters until it copies a newline character, reaches end of file, or copies *n-1* characters. It stores a null character immediately following the last character copied into *s*. **fgets** then increments the associated file pointer by the number of bytes read.

RETURNS

fgets returns *s* if one or more characters are copied and no read error occurs. If it reaches end of file and has not read any characters, or if *pf* does not point at a FILE structure controlling an open file, *s* remains unchanged and **fgets** returns a null pointer. If a read error occurs, the contents of *s* are indeterminate and **fgets** returns a null pointer.

EXAMPLE

To copy a text file, line by line:

```
char buf[80];

ipf = fopen("infile.data", "r");
opf = fopen("outfile.data", "w");
while (fgets(buf, sizeof buf, ipf))
     fputs(buf, opf);
```

SEE ALSO

fputs, **gets**, **puts**

fopen

NAME

fopen - open a file

SYNOPSIS

```
#include <stdio.h>
FILE *fopen(const char *fname, char *type);
```

FUNCTION

fopen tries to open the file with name *fname*. If it succeeds in opening the file, it initializes a FILE structure for operation with the file and connects a stream to it.

The *fname* argument points to a string that contains the name of the file **fopen** is to open. **fopen** makes no distinction between uppercase and lowercase letters in the file name.

Under MVS and MVS/XA, the syntax for *fname* follows the standard TSO rules for qualifying file names. If you enclose the file name in double quotation marks only, the compiler adds your account number to the front of the file name. Your account number is provided by a facility such as **RACF**. For example, if your account number is TS12345, the sequence

```
f = fopen("x.y","r");
```

opens the file **TS12345.X.Y** for reading.

If your system does not support account numbers, nothing is prepended to the file name. If you enclose the file name in single quotation marks and then enclose the result in double quotation marks, the compiler *does not* add your account number to the front of the file name. For example, the sequence

```
f = fopen("'x.y'","r");
```

opens the file **X.Y** for reading.

The notation to specify partitioned datasets is:

```
f = fopen("library.source(mem)","r");
```

This specification accesses the member **mem** from the library **TS12345.LIBRARY.SOURCE**. As with simple sequential files, the specification

```
f = fopen("'ts2331.source(mem)'","r");
```

accesses the member **mem** from the library **TS2331.SOURCE**.

Under VM/CMS, there are two formats for specifying file names. The conventional file specification format is:

fname ftype fmode

where *fname* is the file name, *ftype* is the file type, and *fmode* is the file mode. You may write one or more spaces between components. For example, the file name

```
a b c
```

is equivalent to the file name

a b c

The second format for specifying file names under VM/CMS is

fname.ftype.fmode

where the components have the same meaning as above. The only difference between the two formats is that you use single dots rather than spaces to separate the components of the file name.

In both formats, the file mode is * by default if you omit the file mode specification. If the effective file mode is *, **fopen** searches all accessed minidisks in sequence until:

* When creating or truncating a file, it finds a writable minidisk.
* When opening an existing file, it finds a file with the specified file name and file type.

It is an error to specify a file name and not its corresponding file type.

Under both VM/CMS and MVS, you may use as a file name a DDname that you obtain from a filedef you executed previously, as long as their attributes are compatible. The notation to specify a DDname is

dd:*ddname*

where the string **dd:** is a prefix to the actual DDname *ddname*. Never use single quotation marks around a DDname.

Under both VM/CMS and MVS, use **"*"** to refer to your terminal.

The *type* argument points to a string that must begin with one of the following sequences:

"r" open text file for reading
"w" create text file for writing or truncate old file
"a" append; open text file or create for writing at end of file

"rb" open binary file for reading
"wb" create binary file for writing or truncate old file
"ab" append; open binary file or create for writing at end of file

"r+" open text file for update (reading and writing)
"w+" create text file for update or truncate old file
"a+" append; open text file or create for update, writing at end of file

"r+b" open binary file for update (reading and writing)
"w+b" create binary file for update or truncate old file
"a+b" append; open binary file or create for update, writing at end of file

If **fopen** opens a file for update, you may perform both reads and writes on the stream. However, a read may not directly follow a write, and a write may not directly follow a read without an

intervening **fseek** or **rewind**, unless **fopen** encounters end of file on a read.

If you specify any of the append sequences **"a"**, **"ab"**, **"a+"**, or **"a+b"** to open a file, the C library dynamically forces all subsequent writes to the current end of file, regardless of previous calls to **fseek**. Before each write, the C library repositions the file pointer at the end of the output file and flushes the buffer.

Note that on some file types, a call to **fopen** with *type* **"ab"** or **"a+b"** may cause the C library to write new records beyond the last data previously written, because the file was padded with null characters when it was closed.

In addition to the sequences listed above, *type* may optionally contain one or both of the parameters **recfm** and **lrecl**. Each is followed by an equal sign and a parameter value. The **recfm** parameter specifies the record format. The **lrecl** parameter specifies the logical record length. Use commas to separate the elements of the parameter list. If you specify neither **recfm** or **lrecl**, omit the comma. You may also write spaces before and after the comma separators and equal signs. **fopen** makes no distinction between uppercase and lowercase letters in parameters.

The permissible values for record format specification using **recfm** are **f** or **fixed** for fixed record format and **v** or **variable** for variable length record format.

The logical record length you specify with **lrecl** can be any decimal integer value between 1 and 32768, inclusive. Under VM/CMS, however, if you use a DDname to specify the file to be opened, the logical record length you specify may not exceed 32,760 for fixed record format files, or 32,756 for variable length record format files. The logical record length you specify with **fopen** is *four bytes less* than the logical record length you specify in the filedef for a variable length record format file.

The following example illustrates an **fopen** call to open the file **ABC DEF G** under VM/CMS. The file is a fixed record format file with a record length of 80 bytes. You write this call as

```
f = fopen("abc def g", "r, recfm = f, lrecl = 80");
```

If you are creating a file, any attributes you specify determine the attributes of the new file. If you are truncating an existing file under VM/CMS, and if the file name is not a DDname, then **fopen** behaves as if you are creating a file. If you specify no record format, the default format for a created file is **fixed** for a binary file or **variable** for a text file. If you specify no logical record length, the default value is 80 for a fixed record format file or 255 for a variable length record format file.

If the file you are opening already exists, any attributes you specify must be compatible with those of the existing file. If you specify a record format, it must be the same as for the existing file. If you specify a logical record length, it must be the same value used to create the file.

However, for variable length record format files on VM/CMS, only the length of the longest record in the file is retained by the system. Since the information regarding the original record length with which the file was created may have essentially been lost, **fopen** cannot always enforce this requirement strictly. In addition, the following considerations also apply to variable length record format files on VM/CMS:

* If the file name is not a DDname, and if you specify a logical record length that is larger than the recorded length of the longest record currently in the file, then the value you specify is the logical record length that the C library uses.
* If the file name is not a DDname, and if you specify no logical record length, then the C library uses the larger of 255 and the recorded length of the longest record currently in the file.

Reads and writes behave differently for:

* Text files versus binary files
* Fixed record format files versus variable length record format files
* Replacing records versus appending new records
* Writing short records versus writing long records.

See Chapter 9, "Input/Output" for details on how these factors interact.

RETURNS

fopen returns a pointer to the FILE structure controlling the stream if it opens the file. Otherwise it returns a null pointer.

EXAMPLE

To open a fixed record format binary file for reading:

```
if (!(pf = fopen(infile, "rb, recfm=f, lrecl=80")))
    printf("can't open %s\n", infile);
```

SEE ALSO

fclose, **freopen**, **fseek**, **rewind**

NOTES

Under MVS and MVS/XA, a restriction prohibits opening a member of a partitioned dataset for appending. This restriction is not enforced.

fprintf

NAME

fprintf – write formatted arguments to output stream

SYNOPSIS

```
#include <stdio.h>
int fprintf(FILE *pf, const char *fmt, ...);
```

FUNCTION

fprintf converts a series of arguments specified by its variable length argument list to printable text. It writes this printable text to the file controlled by the FILE structure pointed at by *pf*, under control of a format string pointed at by *fmt*.

The format string consists of literal text to be output, interspersed with "conversion specifications" that determine how **fprintf** interprets arguments and how it will convert them for output. If you specify insufficient arguments for the format string, the results are unpredictable. If **fprintf** exhausts the format string while arguments remain, it ignores the excess arguments. **fprintf** returns when it encounters either the end of the format string or a write error.

Each conversion specification starts with the character **%**. After the **%**, you write the following components, in the order listed:

<**flags**>: zero or more characters which modify the meaning of the conversion specification.

<**field width**>: an optional decimal number which specifies a minimum field width. If the converted value has fewer characters than the field width, **fprintf** pads it on the left or right to the field width. The padding is on the left unless you specify padding on the right with the **-** flag. Spaces are used for padding unless the field width is written with a leading zero digit, in which case zeros are used for padding.

If you write a field width with a leading minus sign, **fprintf** interprets the minus sign as a flag. The field width is always positive.

<**precision**>: a decimal number which specifies the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversion characters; the number of digits to appear after the decimal point for the **e**, **E**, or **f** conversion characters; the maximum number of significant digits for the **g** or **G** conversion characters; or the maximum number of characters to be printed from a string for the **s** conversion character. You write the precision as a period followed by a decimal digit string. **fprintf** treats a null digit string as zero.

h: if present, specifies that the following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a *short* or *unsigned short* argument. **fprintf** type casts the argument value to *short* or *unsigned short* before conversion. It specifies that the following **p** conversion character applies to a "short pointer"

argument. System/370 does not support a short pointer representation. If an **h** appears with any other conversion character, **fprintf** ignores it.

l: if present, specifies that the **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a *long* or *unsigned long* argument. It specifies a "long pointer" or "far pointer" argument if used with the **p** conversion character. System/370 does not support a long pointer representation. If the **l** appears with any other conversion character, **fprintf** ignores it.

L: if present, specifies that the following **e**, **E**, **f**, **g**, or **G** conversion character applies to a *long double* argument. System/370 represents type *long double* the same as *double*. If the **L** appears with any other conversion character, **fprintf** ignores it.

<conversion character>: a character that indicates the type of conversion to be applied.

You may indicate either a field width or precision, or both, by an * instead of a digit string. In this case, the next argument in sequence must be an *int*, which supplies the field width or precision. The arguments supplying field width or precision must appear before the argument to be converted. If you supply a negative value for a precision, **fprintf** behaves as if you left the precision unspecified.

The *<flags>* field consists of zero or more of the following:

space: prepends a space if the first character of the result of a signed conversion is not a sign. **fprintf** ignores this flag if you specify both space and +.

#: converts the result to an "alternate form." For the **c**, **d**, **i**, **s**, and **u** conversion characters, the flag has no effect. For the **o** conversion character, it increases the precision to force the first digit of the result to be zero. For the **x**, **p**, and **X** conversion characters, a nonzero argument value will have **0x** or **0X** prepended to its result. For the **e**, **E**, **f**, **g**, or **G** conversion characters, the result will contain a decimal point, even if no digits follow the point. For the **g** and **G** conversion characters, **fprintf** will not remove trailing zeros from the result, as it normally does.

+: the result of a signed conversion will begin with a plus or minus sign.

−: the result will be left justified within the field.

The *<conversion character>* is one of the following:

%: converts to a **%**. It converts no arguments.

c: converts the least significant byte of the *int* argument to a character.

d, i, o, u, x, X: converts the *int* argument to signed decimal (**d** or **i**), unsigned octal (**o**), unsigned decimal (**u**), or

unsigned hexadecimal notation (**x** or **X**). For a signed decimal, a positive number has no leading sign. Octal has no leading **0**, and hexadecimal has no leading **0X**. **fprintf** uses the letters **abcdef** for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear. If **fprintf** can represent the value being converted in fewer digits, it will add leading zeros to the result. The default precision is 1. The result of converting a zero value with precision of zero is no characters.

e, E: converts the *double* argument to exponential form, such as −5.173e+02. A positive number has no leading sign. There is one digit before the decimal point, and the number of digits after it is equal to the precision. If the precision is not specified, **fprintf** sets it to 6. If the precision is zero, no decimal point appears. **fprintf** rounds the value to the appropriate number of digits. The **E** conversion character will produce a result with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. However, if the exponent in the result is greater than or equal to 100, the result contains additional exponent digits.

f: converts the *double* argument to fraction form, such as −517.297. A positive number has no leading sign. The number of digits following the decimal point is equal to the precision specification. If the precision is not specified, **fprintf** sets it to 6. If the precision is zero, no decimal point appears. If the result contains a decimal point, at least one digit appears before it. **fprintf** rounds the value to the appropriate number of digits.

g, G: converts the *double* argument in style *f* or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style **fprintf** uses depends on the value it converts. It uses style **e** only if the exponent resulting from the conversion is less than −4 or greater than the precision. It removes trailing zeros from the result. A decimal point appears only if it is followed by a digit.

n: stores the number of characters produced so far in the data object designated by the next argument in sequence. The argument must be of type *pointer to int*. It converts no arguments.

p: converts the *pointer to void* argument to a hexadecimal number, using the letters **ABCDEF**. The number of digits in the result is determined by the field width.

s: copies the string pointed at by the *pointer to char* argument to produce the result. Characters are copied up to but not including the terminating null character. If the precision is specified, no more characters are copied other than that value.

If the character after **%** is not valid for a conversion specification, **fprintf** writes it out as literal text. Do not use this feature, however, since future versions of **fprintf** may define additional conversion specifications.

A nonexistent or small field width does not cause truncation of a result. If the result is wider than the specified field width, **fprintf** expands the field width to contain the conversion result.

RETURNS

fprintf returns the number of characters in the conversion. If a write error occurs, or if *pf* does not point at a FILE structure controlling an open file, it returns a negative number.

EXAMPLE

To print an error diagnostic:

```
fprintf(stderr, "%d errors in file %s\n", nerrors, fname);
```

To generate a ten digit octal result with leading zeros:

```
fprintf(stdout, "%.*o\n",  10,  i);
```

SEE ALSO

printf, **sprintf**

NOTES

A call with more conversion specifiers than argument variables will cause unpredictable results.

fputc

NAME

fputc - write a character to output stream

SYNOPSIS

```
#include <stdio.h>
int fputc(int c, FILE *pf);
```

FUNCTION

fputc copies the character *c* to the stream controlled by the FILE structure pointed at by *pf*. **fputc** then increments the associated file pointer by one byte.

RETURNS

fputc returns the value of the character copied to the stream. The character value is type cast to *unsigned char*. If a write error occurs, or if *pf* does not point at a FILE structure controlling an open file, **fputc** returns the value **EOF**.

EXAMPLE

To copy a text file, character by character:

```
FILE *ipf, *opf;

ipf = fopen("infile.data", "r");
opf = fopen("outfile.data", "w");
while ((c = fgetc(ipf)) != EOF)
     fputc(c, opf);
```

SEE ALSO

fgetc, **getc**, **getchar**, **putc**, **putchar**

NOTES

<stdio.h> does not define **fputc** as a macro. **putc** is superior to **fputc** in this regard, even though you must exercise care in calling **putc**.

NAME

fputs – write a text line to output stream

SYNOPSIS

```
#include <stdio.h>
int fputs(const char *s, FILE *pf);
```

FUNCTION

fputs copies characters from the buffer starting at *s* to the stream controlled by the FILE structure pointed at by *pf*. It does not copy the terminating null character. **fputs** increments the associated file pointer by the number of bytes it copies.

RETURNS

fputs returns zero if any characters are written and no write error occurs. **fputs** returns a nonzero value if a write error occurs, or if *pf* does not point at a FILE structure controlling an open file.

EXAMPLE

To copy a text file, line by line:

```
FILE *ipf, *opf;
char buf[80];

ipf = fopen("infile.data", "r");
opf = fopen("outfile.data", "w");
while (fgets(buf, sizeof buf, ipf))
     fputs(buf, opf);
```

SEE ALSO

fgets, gets, puts

fread

NAME

fread - read records from input stream

SYNOPSIS

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nelem, FILE *pf);
```

FUNCTION

fread copies array elements from the stream controlled by the FILE structure pointed at by *pf*. Each element occupies *size* bytes of storage. The first element begins at **(char *)***ptr*. **fread** reads at most *nelem* elements. It then increments the file pointer by the number of bytes actually read.

If a read error occurs or if **fread** reads a partial element, the resulting value of the file pointer is indeterminate. You can use the **ferror** and **feof** functions to distinguish between a read error and end of file.

RETURNS

fread returns the number of whole elements actually read, which may be less than *nelem* if it encounters a read error or end of file. If either *size* or *nelem* is zero, or if *pf* does not point at a FILE structure controlling an open file, **fread** returns zero.

EXAMPLE

To copy a file by records:

```
while (fread(buf, sizeof buf, 1, ipf))
    fwrite(buf, sizeof buf, 1, opf);
```

SEE ALSO

fwrite

NAME

freopen – open a file with existing stream

SYNOPSIS

```
#include <stdio.h>
FILE *freopen(const char *fname, const char *type, FILE *pf);
```

FUNCTION

freopen closes the file controlled by the FILE structure pointed at by *pf*. If the close fails, **freopen** ignores the failure. It then opens the file with name *fname*, initializes the same FILE structure for operation with this file, and connects a stream to it. The *fname* and *type* arguments are the same as for **fopen**.

RETURNS

freopen returns *pf* if the open succeeds. Otherwise it returns a null pointer.

EXAMPLE

To make "xyz.err" the standard error file:

```
if (!freopen("xyz.err", "w", stderr))
     {
     printf("can't set up standard error file\n");
     exit(BAD);
     }
```

SEE ALSO

fclose, fopen

fscanf

NAME

fscanf – read formatted arguments from input stream

SYNOPSIS

```
#include <stdio.h>
int fscanf(FILE *pf, const char *fmt, ...);
```

FUNCTION

fscanf reads input text from the file controlled by the FILE structure pointed at by *pf*, and converts it to values of various types under control of a format string pointed at by *fmt*. It stores these values in data objects designated by a series of pointer arguments specified by its variable length argument list.

The format string consists of literal text to be matched on input, interspersed with "conversion specifications" that determine how **fscanf** interprets arguments and how it will convert values to be stored. If you specify insufficient arguments for the format string, the results are unpredictable. If **fscanf** exhausts the format string while arguments remain, it ignores the excess arguments. **fscanf** returns when it encounters a character it does not expect, the end of the format string, the end of file, or a read error.

The format string may contain:

* Any number of spaces, horizontal tabs, and newline characters which cause input to be read up to the next character that is not whitespace.
* An ordinary character other than **%** which must match the next character of the input stream.

Each conversion specification consists of the character **%**, an optional assignment suppressing character *, an optional maximum field width, an optional **h**, **l**, or **L** indicating the size of the receiving object, and a "conversion specifier."

A conversion specifier directs the conversion of the next input field. **fscanf** places the result in the data object designated by the value of the subsequent pointer argument, unless you specify assignment suppression indicated by a *. Except for the conversion specifiers **c** and **[]**, an input field is a string of characters other than spaces. It extends to the next conflicting character or until **fscanf** exhausts the field width, if specified.

The conversion specifier determines the interpretation of the next input field. If the field does not meet this expectation, **fscanf** returns to its caller. Otherwise, the corresponding pointer argument must be a pointer to the appropriate type. The conversion specifier is one of the following:

%: expect a single **%**. No assignment occurs. If the character after **%** is not a valid conversion character, the behavior is undefined.

c: expect any character. The subsequent argument must be of type *pointer to char*. This conversion specifier suppresses the default **fscanf** behavior of skipping over spaces. To

skip over spaces before obtaining a character, use **%1s** and leave room for a terminating null character. If you specify a field width, the corresponding argument must point at the first element of an array big enough to hold the specified number of characters.

d: expect a decimal integer. The subsequent argument must be of type *pointer to int*. The input format is an optionally signed sequence of decimal digits.

e, f, g: expect a real number. The subsequent argument must be of type *pointer to float*. The field may have an optionally signed sequence of decimal digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e**, followed by an optionally signed sequence of decimal digits.

i: expect an integer. The subsequent argument must be of type *pointer to int*. If the input field begins with the characters **0x** or **0X**, the field is a hexadecimal integer. The rest of the field is a sequence of zero or more hexadecimal digits. If the input field begins with the character **0**, the field is an octal integer. The rest of the field is a sequence of zero or more octal digits. Otherwise, the field is a decimal integer. The rest of the field is an optionally signed sequence of decimal digits.

n: consume no input. The subsequent argument must be of type *pointer to int*. The value that **fscanf** stores is the number of characters read so far on this call to **fscanf**.

o: expect an octal integer. The subsequent argument must be of type *pointer to int*. The field may have an optionally signed sequence of octal digits,

p: expect a pointer. The subsequent argument must be of type *pointer to pointer to void*. The field should be printable text produced by the **%p** conversion of **fprintf**. For any input other than a value produced earlier during the same program execution, the behavior of the **%p** conversion is unpredictable.

s: expect a character string. The subsequent argument must be of type *pointer to char*, and must point to an array large enough to hold the input field plus a terminating null character. **fscanf** adds the terminating null character automatically. The input field terminates before a space, horizontal tab, or newline.

u: expect an unsigned decimal integer. The subsequent argument must be of type *pointer to int*. The input format is an optionally signed sequence of decimal digits.

x: expect a hexadecimal integer. The subsequent argument must be of type *pointer to int*. The field may have an optionally signed sequence of hexadecimal digits.

[: expect a string that is not delimited by spaces. The subsequent argument must be of type *pointer to char*, just

as for **%s**. Follow the left bracket with a set of characters and a right bracket. The characters between the brackets define the set of characters making up the string. If the first character is not a circumflex ^, the input field consists of all characters up to the first character that is *not* in the set between the brackets. If the first character after the left bracket is a circumflex, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. After copying the input field, **fscanf** then stores a null character following the last character copied.

You may precede the conversion characters **d**, **i**, **o**, **u**, and **x** with **l** to indicate that the subsequent argument is of type *pointer to long* rather than *pointer to int*, or by **h** to indicate that it is of type *pointer to short*. Similarly, you may precede the conversion characters **e** and **f** with **l** to indicate that the subsequent argument is of type *pointer to double* rather than *pointer to float*, or by **L** to indicate type *pointer to long double*.

You may write **E** for **e**, **G** for **g**, or **X** for **x**. This has no effect on the conversion performed, however. For either form, **fscanf** accepts both uppercase and lowercase letters in the input field.

If conversion terminates on an unexpected input character, **fscanf** leaves that character unread in the input stream. It calls the function **ungetc** to push the character back into the input stream. It will leave trailing whitespace, including a newline, unread unless matched in the format string. The success of literal matches and suppressed assignments is not directly determinable except via the **%n** conversion specifier.

RETURNS

fscanf returns the number of input fields converted and assigned. This number can be zero if **fscanf** encounters unexpected input before the first field is converted and assigned. **fscanf** returns the value **EOF** if it encounters end of file before the first unexpected input or conversion and assignment, or if *pf* does not point at a FILE structure controlling an open file.

If a range error occurs, **fscanf** sets **errno** to **ERANGE** and stores a special value in the data object. For a floating type, this special value is **HUGE_VAL** for positive values too large to represent, or **-HUGE_VAL** for negative values too large to represent. For integer types, it is the largest representable value of the appropriate sign.

EXAMPLE

To read a hexadecimal number into a *long*:

```
long hexlong;

fscanf(infile, "%8Lx", &hexlong);
```

SEE ALSO

scanf, **sscanf**, **ungetc**

NAME

fseek - set file pointer

SYNOPSIS

```
#include <stdio.h>
int fseek(FILE *pf, long loff, int sense);
```

FUNCTION

fseek sets the file pointer for the stream controlled by the FILE structure pointed at by *pf*.

For a binary stream connected to a fixed record format file, the file pointer is set to the signed distance in bytes that *loff* specifies from the position that *sense* specifies. The specified position is

* The beginning of the file if *sense* has the value **SEEK_SET**
* The current position if *sense* has the value **SEEK_CUR**
* The end of the file if *sense* has the value **SEEK_END**.

No other value of *sense* is valid. The C library may append one or more null characters to the end of a fixed record format file. Remember this if you call **fseek** with *sense* equal to the value **SEEK_END**.

For *any* stream, including a text stream or a binary stream connected to a variable length record format file, you may specify either:

* An offset of zero, and any of the values for *sense* described above
* An offset equal to the value of a file pointer returned by a call to **ftell**, with *sense* equal to the value **SEEK_SET.**

The first choice sets the file pointer just as for a binary stream. The second choice sets the file pointer to the same position it had when you obtained the value by calling **ftell**. Use only values returned for the same stream. Do not close the file controlled by that stream between the call to **ftell** and the call to **fseek**.

The return value from **ftell** is valid only if

* The current file pointer is within the first 32,768 bytes of the start of a physical record in the file
* The physical record is one of the first 131,072 records of the file.

ftell returns -1 for all other file positions, which is not an acceptable value of *loff* for **fseek**.

fseek eliminates any effects of an earlier call to **ungetc**, if you have not read the character pushed back in the interim. After an **fseek** call, the next operation on a stream opened for update may be either input or output.

RETURNS

fseek returns a nonzero value for invalid arguments if the file cannot support the requested change in file pointer, or if *pf* does

not point at a FILE structure controlling an open file.

EXAMPLE

To go to the record indicated by **index** in a binary file:

```
fseek(pf, (long)(index * sizeof (struct record)), SEEK_SET);
```

SEE ALSO

fopen, **ftell**, **rewind**, **ungetc**

NOTES

A system restriction prohibits performing seek operations with VBS (Variable Block Span) files under MVS and MVS/XA.

A restriction prohibits seeking on a member of a partitioned dataset which is opened for writing, updating, or appending under MVS and MVS/XA.

NAME

ftell - get file pointer

SYNOPSIS

```
#include <stdio.h>
long ftell(FILE *pf);
```

FUNCTION

ftell gets the current value of the file pointer for the stream controlled by the FILE structure pointed at by *pf*. You can use this value in a later call to **fseek** to return the file pointer to its position at the time of the **ftell** call.

For a binary stream connected to a fixed record format file, the value of the file pointer is the offset in bytes from the beginning of the file. For a text file or variable length record format binary file, the file pointer is an encoded value. This value is valid if

* The current file position is within the first 32,768 bytes of the start of a physical record in the file
* The physical record is one of the first 131,072 records of the file

ftell may return -1 for any other file positions, which is not an acceptable value for use on a later call to **fseek**. Note that a file positioned at byte 32768 of record 131072 may also cause **ftell** to return -1.

RETURNS

ftell returns the current value of the file pointer. The value is -1 if it cannot encode the file position in the file pointer, or if *pf* does not point at a FILE structure controlling an open file.

EXAMPLE

To save the current position in a text file to return to later:

```
fprintf(pf, "total: xxxxx\n");
marker = ftell(pf);
.....
fseek(pf, marker, SEEK_SET);
fprintf(pf, "total: %5i\n", total);
```

SEE ALSO

fseek, **rewind**

NOTES

The difference between two file pointers is meaningful only for a stream connected to a fixed record format file. In this case, the difference measures the number of bytes in the file between the two file positions. For text files, variable length record format files on VM/CMS, and all file types on MVS and MVS/XA except F, FA, FBS, and FBSA, the file pointer is an encoded value.

fwrite

NAME

fwrite – write records to output stream

SYNOPSIS

```
#include <stdio.h>
size_t fwrite(const void *ptr,     \
size_t size, size_t nelem, FILE *pf);
```

FUNCTION

fwrite copies array elements to the stream controlled by the FILE structure pointed at by *pf*. Each element occupies *size* bytes of storage. The first element begins at **(char *)***ptr*. **fwrite** copies *nelem* elements. It then increments the file pointer by the number of bytes written.

If a write error occurs, the resulting value of the file pointer is unpredictable.

RETURNS

fwrite returns the number of whole items actually written, which may be less than *nelem* if a write error occurs. If either *size* or *nelem* is zero, or if *pf* does not point at a FILE structure controlling an open file, **fwrite** returns zero.

EXAMPLE

To copy a file record by record:

```
while (fread(buf, sizeof buf, 1, ipf))
     fwrite(buf, sizeof buf, 1, opf);
```

SEE ALSO

fread

NAME

getc – read a character from input stream

SYNOPSIS

```
#include <stdio.h>
int getc(FILE *pf);
```

FUNCTION

getc obtains the next input character, if any, from the stream controlled by the FILE structure pointed at by *pf*. **getc** then increments the associated file pointer by one byte.

<stdio.h> defines **getc** as a macro that expands to an expression that evaluates *pf* more than once. If you must write this argument with side effects, call the equivalent function **fgetc** or use the **#undef** preprocessor directive to remove any macro definition for the identifier **getc**.

RETURNS

getc returns the value of the next character from the input stream pointed at by *pf*. The character value is type cast to *unsigned char*. If **getc** encounters end of file or a read error, or if *pf* does not point at a FILE structure controlling an open file, it returns the value **EOF**.

EXAMPLE

To copy a text file, character by character:

```
FILE *ipf, *opf;

ipf = fopen("infile.data", "r");
opf = fopen("outfile.data", "w");
while ((c = getc(ipf)) != EOF)
     putc(c, opf);
```

SEE ALSO

fgetc, **fputc**, **getchar**, **putc**, **putchar**

getchar

NAME

getchar - read a character from standard input

SYNOPSIS

```
#include <stdio.h>
int getchar();
```

FUNCTION

getchar obtains the next input character, if any, from the standard input stream. **getchar** then increments the associated file pointer by one byte.

RETURNS

getchar returns the value of the next character from the standard input stream. The character value is type cast to *unsigned char*. If **getchar** encounters end of file or a read error, or if **stdin** does not control an open file, it returns the value **EOF**.

EXAMPLE

To copy a text file, character by character:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

SEE ALSO

fgetc, **fputc**, **getc**, **putc**, **putchar**

NAME

gets - read a text line from standard input

SYNOPSIS

```
#include <stdio.h>
char *gets(char *s);
```

FUNCTION

gets copies characters, from the standard input stream, to the buffer starting at *s*. **gets** copies characters until it encounters a newline character or reaches end of file. If it encounters a newline character, it discards it and stores a null character immediately following the last character copied into *s*.

RETURNS

gets returns *s* if any characters are copied and no read error occurs. If it reaches end of file before copying any characters, or if **stdin** does not control an open file, **gets** returns a null pointer. If a read error occurs, the contents of *s* are unpredictable and **gets** returns a null pointer.

EXAMPLE

To copy standard input to standard output:

```
while (gets(buf) &&
        !puts(buf))
    ;
```

SEE ALSO

fgets, **fputs**, **puts**

NOTES

There is no definite limit on the size of the line that **gets** reads. **fgets** is superior to **gets** in this regard.

perror

NAME

perror - map error number

SYNOPSIS

```
#include <stdio.h>
const char *perror(const char *s);
```

FUNCTION

perror maps the error number stored in **errno** to an error message. **perror** provides informative error messages for all of the values that the C library stores in **errno**. **errno** may have any stored value. If *s* is not a null pointer, and **errno** is nonzero, **perror** writes a line to the standard error stream consisting of the string that *s* points to, a colon, a space, the error message, and a newline.

If *s* is a null pointer, **perror** returns a pointer to the error message string and performs no output.

RETURNS

perror returns a pointer to the error message string if *s* is a null pointer. Otherwise **perror** returns a null pointer.

EXAMPLE

The statement

```
if (errno)
    perror("quadrature function");
```

might produce the output

```
quadrature function: range error
```

SEE ALSO

<math.h>, **<stdefs.h>**

NAME

printf – write formatted arguments to standard output

SYNOPSIS

```
#include <stdio.h>
int printf(const char *fmt, ...);
```

FUNCTION

printf converts a series of arguments specified by its variable length argument list to printable text. It writes this printable text to the standard output stream, under control of a format string pointed at by *fmt*. Its behavior is entirely equivalent to the call

```
fprintf(stdout, fmt, ...)
```

with the same variable length argument list.

RETURNS

printf returns the number of characters in the conversion. If a write error occurs, or if **stdout** does not control an open file, it returns a negative number.

EXAMPLE

To print **arg**, which has type *double* and the decimal value 5100.53:

```
printf("%8.2f\n", arg);
printf("%*.*f\n", 8, 2, arg);
```

Both forms will output:

```
' 5100.53'
```

Note that the first character of the above output is a blank space.

SEE ALSO

fprintf, **sprintf**

putc

NAME

putc – write a character to output stream

SYNOPSIS

```
#include <stdio.h>
int putc(int c, FILE *pf);
```

FUNCTION

putc copies the character *c* to the stream controlled by the FILE structure pointed at by *pf*. The file pointer associated with that stream determines where in the file that *c* is copied. **putc** then increments the associated file pointer by one byte.

<stdio.h> defines **putc** as a macro that expands to an expression that evaluates *pf* more than once. If you must write this argument with side effects, call the equivalent function **fputc** or the **#undef** preprocessor directive to remove any macro definition for the identifier **putc**.

RETURNS

putc returns the value of the character copied to the stream. The character value is type cast to *unsigned char*. If a write error occurs, or if *pf* does not point at a FILE structure controlling an open file, **putc** returns the value **EOF**.

EXAMPLE

To copy a text file, character by character:

```
FILE *ipf, *opf;

ipf = fopen("infile.data", "r");
opf = fopen("outfile.data", "w");
while ((c = getc(ipf)) != EOF)
     putc(c, opf);
```

SEE ALSO

fgetc, **fputc**, **getc**, **getchar**, **putchar**

NAME

putchar – write a character to standard output

SYNOPSIS

```
#include <stdio.h>
int putchar(int c);
```

FUNCTION

putchar copies the character *c* to the standard output stream. The file pointer associated with that stream determines where in the file that *c* is copied. **putchar** then increments the associated file pointer by one byte.

RETURNS

putchar returns the value of the character copied to the stream. The character value is type cast to *unsigned char*. If a write error occurs, or if **stdout** does not control an open file, **putchar** returns the value **EOF**.

EXAMPLE

To copy a text file, character by character:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

SEE ALSO

fgetc, **fputc**, **getc**, **getchar**, **putc**

puts

NAME

puts – write a text line to standard output

SYNOPSIS

```
#include <stdio.h>
int puts(const char *s);
```

FUNCTION

puts copies characters from the string starting at *s* to the standard output stream. It does not copy the terminating null character. It then writes a newline character to the stream.

RETURNS

puts returns zero if any characters are written and no write error occurs. **puts** returns a nonzero value if a write error occurs, or if **stdout** does not control an open file.

EXAMPLE

To copy the standard input to the standard output:

```
while (gets(buf) &&
        !puts(buf))
    ;
```

SEE ALSO

fgets, **fputs**, **gets**

NAME

remove – remove a file

SYNOPSIS

```
#include <stdio.h>
int remove(const char *fname);
```

FUNCTION

remove removes the file with name *fname*. You cannot open the file after you remove it. For files with cataloged MVS names or CMS filenames, you may create a new file with the same name after you remove a file. Files accessed using a data definition (a DD statement under MVS or a FILEDEF under VM/CMS) cannot be removed. An attempt to reopen a data definition name file with invalid file attributes will fail.

The file must not be open when you remove it.

RETURNS

remove returns a nonzero value if the file does not exist or if **remove** cannot remove it.

EXAMPLE

To use and remove a file:

```
pf = fopen("x.y", "w");
process(pf);
if (remove("x.y"))
     printf("can't remove file\n");
```

SEE ALSO

rename, **tmpnam**

rename

NAME

rename - change file name

SYNOPSIS

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

FUNCTION

rename changes the name of the file with name *old* to the name *new*. If it is necessary to copy the file to change the name, **rename** will fail.

The file must not be open when you rename it.

RETURNS

Under VM/CMS, **rename** returns a nonzero value if the file does not exist or if it cannot rename the file. Under MVS, **rename** is not implemented and always returns failure.

EXAMPLE

To rename a test file:

```
if (rename(otest, ntest))
     printf("can't rename %s\n", otest);
```

SEE ALSO

remove

NAME

rewind – set file pointer to beginning of file

SYNOPSIS

```
#include <stdio.h>
void rewind(FILE *pf);
```

FUNCTION

rewind sets the file pointer for the stream controlled by the FILE structure pointed at by *pf* to the beginning of the file. It also resets the end of file and error indicators.

The function call **rewind(pf)** is equivalent to the expression

```
(void) fseek(pf, 0L, SEEK_SET)
```

RETURNS

Nothing.

EXAMPLE

To change modes from writing to reading:

```
while (fill_up(s) && !fputs(s, pf))
     ;
rewind(pf);
while (fgets(s, BUFSIZ, pf))
     empty(s);
```

SEE ALSO

fseek

scanf

NAME

scanf – read formatted arguments from standard input

SYNOPSIS

```
#include <stdio.h>
int scanf(const char *fmt, ...);
```

FUNCTION

scanf reads input text from the standard input stream, and converts it to values of various types under control of a format string pointed at by *fmt*. It stores these values in data objects designated by a series of pointer arguments specified by its variable length argument list. Its behavior is entirely equivalent to the call

```
fscanf(stdin, fmt, ...)
```

with the same variable length argument list.

RETURNS

scanf returns the number of input fields converted and assigned. This number can be zero if **scanf** encounters unexpected input before the first field is converted and assigned. **scanf** returns the value **EOF** if it encounters end of file before the first unexpected input or conversion and assignment, or if **stdin** does not control an open file.

EXAMPLE

To emit a warning in response to a request:

```
printf("are you sure? ");
fflush(stdout);
if (scanf("%c", &ans) && (ans == 'y' || ans == 'Y'))
     return (YES);
```

SEE ALSO

fscanf, **sscanf**

NAME

setbuf - set stream buffer

SYNOPSIS

```
#include <stdio.h>
void setbuf(FILE *pf, char *buf);
```

FUNCTION

setbuf causes the stream controlled by the FILE structure pointed at by *pf* to use as its buffer the buffer pointed at by *buf*. If you call **setbuf** after a file is opened and before you call a function that reads or writes the stream connected to the open file, then no buffer will be dynamically allocated on the first read or write to the stream. If you call **setbuf** for a stream after the first read or write to the stream, any output is flushed from the current buffer. If the current buffer is dynamically allocated by the C library, it is deallocated.

You must provide a buffer that occupies at least **BUFSIZ** bytes. Storage for the buffer must not be deallocated until after the file is closed. If the buffer has dynamic lifetime, close the file before control leaves the block that allocates the buffer. Once you close the file, the buffer disappears. It will not be reused if the stream that controlled the file is used when another file is opened.

Do not store into the buffer while the file is open. The C library may choose not to use the buffer you provide, or to use it in ways you cannot anticipate. Do not access the buffer while the file is open.

RETURNS

Nothing.

EXAMPLE

To force **stdout** to use a preferred buffer:

```
static char mybuf[BUFSIZ];

setbuf(stdout, mybuf);
```

SEE ALSO

fopen, **freopen**, **fclose**, **setvbuf**

NOTES

setbuf is an older function that performs just one of the services offered by **setvbuf**. **setvbuf** is superior to **setbuf** in this regard.

setvbuf

NAME

setvbuf – set stream buffering strategy

SYNOPSIS

```
#include <stdio.h>
int setvbuf(FILE *pf, char *buf, int type, int n);
```

FUNCTION

setvbuf sets the buffering strategy to the type of strategy you specify for the stream controlled by the FILE structure pointed at by *pf*. *type* must have one of the following values:

_IOFBF – to specify full buffering. The buffer is flushed only when it is full, when you call **fflush**, or when you close the file.

_IOLBF – to specify line buffering. The buffer is flushed when the buffer is full, when you write a newline character to the stream, when you read from the stream, or when you close the file.

_IONBF – to specify no buffering. The buffer is flushed whenever you write to the stream.

If *pf* is not a null pointer it also causes the stream to use as its buffer the buffer pointed at by *buf*. The length of the buffer is *n*. If you call **setvbuf** after a file is opened and before you call a function that reads or writes the stream connected to the open file, then no buffer will be dynamically allocated on the first read or write to the stream. If you call **setvbuf** for a stream after the first read or write to the stream, **setvbuf** sets the read/write error indicator.

If you specify a buffer, storage for the buffer must not be deallocated until after the file is closed. If the buffer has dynamic lifetime, close the file before control leaves the block that allocates the buffer. Once you close the file, the buffer disappears. It will not be reused if the stream that controlled the file is used when another file is opened.

Do not store into the buffer while the file is open. The C library may choose not to use the buffer you provide, or to use it in ways you cannot anticipate. Do not access the buffer while the file is open.

RETURNS

setvbuf returns a nonzero value if you specify invalid arguments, or if *pf* does not point at a FILE structure controlling an open file.

EXAMPLE

To force **stdout** to use a preferred buffer:

```
static char mybuf[4096];

setvbuf(stdout, mybuf, _IOFBF, sizeof mybuf);
```

SEE ALSO

fopen, freopen, fclose, setbuf

sprintf

NAME

sprintf – write formatted arguments to a string

SYNOPSIS

```
#include <stdio.h>
int sprintf(char *s, const char *fmt, ...);
```

FUNCTION

sprintf converts a series of arguments specified by its variable length argument list to printable text. It stores this printable text in the buffer pointed at by *s*, under control of a format string pointed at by *fmt*. Its behavior is otherwise the same as that of **fprintf**, called with the same variable length argument list.

sprintf stores a null character immediately after the last character stored in the buffer. You must provide a buffer large enough to store all of these characters.

RETURNS

sprintf returns the number of characters in the conversion. This does not include the null character.

EXAMPLE

To center a title:

```
extern int width;
int n;
n = sprintf(buf, "%i %s, Year %i", day,          \
                mon_str[mo], year);
printf("%*s\n", (width + n) / 2, buf);
```

SEE ALSO

fprintf, **printf**

NAME

sscanf - read formatted arguments from a string

SYNOPSIS

```
#include <stdio.h>
int sscanf(char *s, const char *fmt, ...);
```

FUNCTION

sscanf obtains input text from the string pointed at by *s*, and converts it to values of various types under control of a format string pointed at by *fmt*. It stores these values in data objects designated by a series of pointer arguments specified by its variable length argument list. Its behavior is otherwise the same as that of **fscanf**, called with the same variable length argument list.

RETURNS

sscanf returns the number of input fields converted and assigned. This number can be zero if **sscanf** encounters unexpected input before the first field is converted and assigned. **sscanf** returns the value **EOF** if it encounters the terminating null character for the string before the first unexpected input or conversion and assignment.

EXAMPLE

To expand tabs before converting input text:

```
while (fgets(buf, sizeof buf, stdin))
     {
     expand(buf, tab_stops);
     n = sscanf(buf, fmt, &v1, &v2, &v3);
     process(n, &v1, &v2, &v3);
     }
```

SEE ALSO

fscanf, **scanf**

tmpfile

NAME

tmpfile – create temporary file

SYNOPSIS

```
#include <stdio.h>
FILE *tmpfile();
```

FUNCTION

tmpfile creates a temporary file, opens it as a binary stream in update mode, initializes a FILE structure for operation with the file, and connects a stream to it. The C library automatically removes a temporary file when you close the file or when your program terminates.

RETURNS

tmpfile returns a pointer to the FILE structure controlling the stream if it opens the file. Otherwise it returns a null pointer.

EXAMPLE

To create and use a temporary file:

```
pf = tmpfile();
pass1(pf);
rewind(pf);
pass2(pf);
```

SEE ALSO

tmpnam

NAME

tmpnam – generate temporary file name

SYNOPSIS

```
#include <stdio.h>
char *tmpnam(char *s);
```

FUNCTION

tmpnam creates a file name that you can use as the name of a temporary file, and stores the name as a string in memory. It chooses a file name which is likely not to conflict with normal user file names. **tmpnam** generates a different name each time you call it. It will generate a minimum of **TMP_MAX** unique names.

The format of the file name is *$nnnnnnn.CTEMP* under CMS, where *nnnnnnn* is a sequentially assigned integer value. Under MVS and MVS/XA, **tmpnam** returns a DDname corresponding to a unique system generated dataset name. The dataset is always deleted at the end of your batch job or terminal session. See *OS/VS2 MVS JCL* (GC28-0962) for more information on system generated temporary dataset names.

Call **fopen** or **freopen** with the file name to open the file. Call **fclose** and **remove** to remove the file before your program terminates.

If *s* is a null pointer, **tmpnam** stores the temporary file name in a static buffer. Otherwise, *s* must be a buffer whose length is at least **L_tmpnam** bytes. **tmpnam** stores the temporary file name into this buffer.

RETURNS

tmpnam returns a pointer to the buffer where it stores the file name.

EXAMPLE

To obtain a set of temporary file names:

```
for (i = 0; i < NFILES; ++i)
     tmpnam(&temp_file[i]);
```

SEE ALSO

tmpfile

NOTES

If you call **tmpnam** again, it may overwrite its static buffer. Make sure you copy out the name, or use it as often as you need, before calling **tmpnam** again.

Under VM/CMS, if you invoke multiple C programs in the same environment, both may obtain the same name from **tmpnam**.

ungetc

NAME

ungetc - push character back into input stream

SYNOPSIS

```
#include <stdio.h>
int ungetc(int c, FILE *pf);
```

FUNCTION

ungetc pushes back the character *c* into the stream controlled by the FILE structure pointed at by *pf*. The character is retained until you read it or until you call **fseek** or **rewind** on the stream. It is never written to the file. You need not push back the same character you read.

The C library supports only one character of push back for each stream. Remember that the functions **fscanf** and **scanf** may call **ungetc** just before they return.

RETURNS

ungetc returns the value of the character if it can push the character back. The character value is type cast to *unsigned char*. If it cannot push back the character, or if *pf* does not point at a FILE structure controlling an open file, **ungetc** returns the value **EOF**.

EXAMPLE

To read an identifier:

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

main()
     {
     unsigned char ident[30], c;
     int i;

     if (isalpha(c = getchar()))
          for (i = 0, ident[0] = c;    \
                    (c = getchar()) != EOF;)
               if (!(isalpha(c) || isdigit(c) || c == '_'))
                    {
                    ungetc(c, stdin);
                    break;
                    }
               else if (++i < sizeof (ident))
                    ident[i] = c;
     ident[++i] = '\0';
     printf("identifier is: %s\n", ident);
     }
```

SEE ALSO

fgetc, fseek, getc, getchar, rewind

NAME

vfprintf – write formatted argument list to output stream

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>
int vfprintf(FILE *pf, const char *fmt, va_list arg);
```

FUNCTION

vfprintf converts a series of arguments specified by a variable length argument list to printable text. It writes this printable text to the stream controlled by the **FILE** structure pointed at by *pf*, under control of a format string pointed at by *fmt*. You must store information on the variable length argument list in *arg*, by expanding the macro **va_start**. Do this in the function that contains the variable length argument list.

vfprintf otherwise behaves the same as **fprintf** called directly with the same variable length argument list.

RETURNS

vfprintf returns the number of characters in the conversion. If a write error occurs, or if *pf* does not point at a FILE structure controlling an open file, it returns a negative number.

EXAMPLE

To write to two streams at once:

```
void twoprint(char *fmt, ...)
     {
     va_list ap;

     va_start(ap, fmt);
     vfprintf(stderr, fmt, ap);
     va_end(ap);

     va_start(ap, fmt);
     vprintf(fmt, ap);
     va_end(ap);
     }
```

SEE ALSO

<stdarg.h>, fprintf, vprintf, vsprintf

vprintf

NAME

vprintf – write formatted argument list to standard output

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>
int vprintf(const char *fmt, va_list arg);
```

FUNCTION

vprintf converts a series of arguments specified by a variable length argument list to printable text. It writes this printable text to the standard output stream, under control of a format string pointed at by *fmt*. You must store information on the variable length argument list in *arg*, by expanding the macro **va_start**. Do this in the function that contains the variable length argument list.

vprintf otherwise behaves the same as **printf** called directly with the same variable length argument list.

RETURNS

vprintf returns the number of characters in the conversion. If a write error occurs, or if **stdout** does not control an open file, it returns a negative number.

EXAMPLE

To write to two streams at once:

```
void twoprint(char *fmt, ...)
     {
     va_list ap;

     va_start(ap, fmt);
     vfprintf(stderr, fmt, ap);
     va_end(ap);

     va_start(ap, fmt);
     vprintf(fmt, ap);
     va_end(ap);
     }
```

SEE ALSO

<stdarg.h>, **fprintf**, **vfprintf**, **vsprintf**

NAME

vsprintf - write formatted argument list to a string

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>
int vsprintf(char *s, const char *fmt, va_list arg);
```

FUNCTION

vsprintf converts a series of arguments specified by a variable length argument list to printable text. It stores this printable text in the buffer pointed at by *s*, under control of a format string pointed at by *fmt*. You must store information on the variable length argument list in *arg*, by expanding the macro **va_start**. Do this in the function that contains the variable length argument list.

vprintf otherwise behaves the same as **sprintf** called directly with the same variable length argument list.

RETURNS

sprintf returns the number of characters in the conversion.

EXAMPLE

To center all output:

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

main()
     {
     void center_print();

     center_print(80, "Annual Report");
     }

void center_print(int width, char *fmt, ...)
     {
     va_list ap;
     char buf[133];
     int n;

     va_start(ap, fmt);
     n = vsprintf(buf, fmt, ap);
     printf("%*s\n", (width + n) / 2, buf);
     va_end(ap);
     }
```

SEE ALSO

<stdarg.h>, fprintf, vfprintf, vprintf

stdlib.h

NAME

stdlib.h – header file for general utilities

SYNOPSIS

```
#include <stdlib.h>
```

FUNCTION

The header file **<stdlib.h>** declares functions that may be used to perform a variety of services, including: storage allocation, conversion of printable text to arithmetic types, and program termination.

<stdlib.h> defines the following type:

onexit_t – a scalar type used as the argument type and return value type for **onexit**.

<stdlib.h> declares the following functions:

abort – terminate program execution abnormally.

atof – convert buffer to floating.

atoi – convert buffer to integer.

atol – convert buffer to long integer.

calloc – allocate and clear storage.

exit – terminate program execution.

free – free storage.

getenv – get environment variable.

malloc – allocate storage.

onexit – call function on program termination.

rand – generate pseudo random number.

realloc – reallocate storage.

srand – seed pseudo random number generator.

strtod – convert buffer to floating with error checking.

strtol – convert buffer to long integer with error checking.

system – execute a command.

NAME

abort – terminate program execution abnormally

SYNOPSIS

```
#include <stdlib.h>
void abort();
```

FUNCTION

abort reports the signal **SIGABRT** by calling **kill**. If **kill** returns, **abort** then calls **exit** with a nonzero argument to indicate unsuccessful termination.

RETURNS

abort never returns to its caller.

EXAMPLE

To terminate if a routine cannot access a file:

```
if (!(fp = fopen(DBMASTER, "r+b")))
     {
     printf("can't open database index master\n");
     abort();
     }
```

SEE ALSO

assert, **exit**, **kill**, **onexit**, **signal**

atof

NAME

atof – convert buffer to floating

SYNOPSIS

```
#include <stdlib.h>
double atof(const char *nptr);
```

FUNCTION

atof converts the string at *nptr* to a value of type *double*. It interprets the string as the text representation of a real number. The string may have leading whitespace, an optionally signed sequence of decimal digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e** followed by an optionally signed sequence of decimal digits. The text representation ends just before the first character that does not satisfy this format.

RETURNS

atof returns the converted *double* value. It returns zero if no characters are converted.

EXAMPLE

To read a string and convert it as a floating constant:

```
fgets(buf, sizeof buf, stdin);
d = atof(buf);
```

SEE ALSO

atoi, **atol**, **strtod**, **strtol**

NOTES

atof does not check for overflow or underflow. It provides no way to determine how many characters are converted. The function **strtod** is superior to **atof** in this regard.

NAME

atoi – convert buffer to integer

SYNOPSIS

```
#include <stdlib.h>
int atoi(const char *nptr);
```

FUNCTION

atoi converts the string at *nptr* to a value of type *int*. It interprets the string as the text representation of an integer. The string may have leading whitespace, followed by an optionally signed sequence of decimal digits. The text representation ends just before the first character that does not satisfy this format.

RETURNS

atoi returns the converted *int* value. It returns zero if no characters are converted.

EXAMPLE

To read a string and convert it as a integer constant:

```
fgets(buf, sizeof buf, stdin);
i = atoi(buf);
```

SEE ALSO

atof, **atol**, **strtod**, **strtol**

NOTES

atoi does not check for overflow. It provides no way to determine how many characters are converted. The function **strtol** is superior to **atoi** in this regard.

atol

NAME

atol – convert buffer to long integer

SYNOPSIS

```
#include <stdlib.h>
int atol(const char *nptr);
```

FUNCTION

atol converts the string at *nptr* to a value of type *long*. It interprets the string as the text representation of an integer. The string may have leading whitespace, followed by an optionally signed sequence of decimal digits. The text representation ends just before the first character that does not satisfy this format.

RETURNS

atol returns the converted *long* value. It returns zero if no characters are converted.

EXAMPLE

To read a string and convert it as a *long* constant:

```
fgets(buf, sizeof buf, stdin);
i = atol(buf);
```

SEE ALSO

atof, **atoi**, **strtod**, **strtol**

NOTES

atol does not check for overflow. It provides no way to determine how many characters are converted. The function **strtol** is superior to **atol** in this regard.

NAME

calloc - allocate and clear storage

SYNOPSIS

```
#include <stdlib.h>
void *calloc(size_t nelem, size_t elsize);
```

FUNCTION

calloc allocates storage for a data object whose size in bytes is the product of *nelem* and *elsize*. It then stores a null character in each of these bytes. The allocated storage is aligned on a storage boundary acceptable for any data object type.

RETURNS

calloc returns a pointer to the allocated storage, if there is enough storage available. Otherwise it returns a null pointer.

EXAMPLE

To allocate and set to zero an array of ten doubles:

```
double *pd;

pd = calloc(10, sizeof *pd);
```

SEE ALSO

free, **malloc**, **realloc**

NAME

exit - terminate program execution

SYNOPSIS

```
#include <stdlib.h>
void exit(int status);
```

FUNCTION

exit calls all functions registered with **onexit** in reverse order of registry, closes all files, removes temporary files, and terminates program execution. You call **exit** with a *status* value of zero to indicate successful termination. Any nonzero value indicates some form of unsuccessful termination.

RETURNS

exit will never return to its caller.

EXAMPLE

To exit with the correct status:

```
exit(nerrors != 0);
```

SEE ALSO

abort, **onexit**

NAME

free – free storage

SYNOPSIS

```
#include <stdlib.h>
void free(void *ptr);
```

FUNCTION

free deallocates storage allocated on an earlier call to **calloc**, **malloc**, or **realloc**. If *ptr* is a null pointer, **free** does nothing.

You free allocated data objects to make more storage available for later allocation.

RETURNS

Nothing.

You must not make use of *ptr* after you call **free**.

EXAMPLE

To give back an allocated data object:

```
free(pd);
```

SEE ALSO

calloc, **malloc**, **realloc**

NOTES

Call **free** only with a null pointer or with a pointer value returned earlier by **calloc**, **malloc**, or **realloc**.

getenv

NAME

getenv – get environment variable

SYNOPSIS

```
#include <stdlib.h>
char *getenv(const char *name);
```

FUNCTION

getenv searches an externally supplied environment list for a string of the form *"NAME=value"*. If the string at *name* compares equal to a *NAME* in the environment list, **getenv** returns a pointer to the corresponding *value* string. Do not store into the string.

Under CMS, you can use the **GLOBALV** command to define global variables to be used in a C program. These global variables must be defined under the group name **CENV**. You use **getenv** to access these global variables.

Under MVS, the environment list is empty.

RETURNS

getenv returns a pointer to the start of the *value* string if *name* compares equal to *NAME* for that string. Otherwise **getenv** returns a null pointer.

EXAMPLE

To check the environment for the global variable **MYVAR**:

```
char *r;
char *s = "MYVAR";

if ((r = getenv(s)) == NULL)
     printf("GLOBAL variable MYVAR not found\n");
else
     printf("GLOBAL variable MYVAR defined as: %s\n", r);
```

NAME

malloc – allocate storage

SYNOPSIS

```
#include <stdlib.h>
void *malloc(size_t nbytes);
```

FUNCTION

malloc allocates storage for a data object whose size in bytes is *nbytes*. The allocated storage is aligned on a storage boundary acceptable for any data object type.

RETURNS

malloc returns a pointer to the allocated storage, if there is enough storage available. Otherwise it returns a null pointer.

EXAMPLE

To allocate an array of ten doubles:

```
double *pd;

pd = malloc(10 * sizeof *pd);
```

SEE ALSO

calloc, free, realloc

onexit

NAME

onexit - call function on program termination

SYNOPSIS

```
#include <stdlib.h>
onexit_t onexit(onexit_t (*pfunc)(void));
```

FUNCTION

onexit stores the function pointer *pfunc* in a static data object internal to the C library. It returns the previous value stored in the data object.

When your program calls **exit**, **exit** calls the function pointed at by the value stored in the data object. It calls the function before it closes any files or removes any temporary files. Since **exit** is called when your program returns from **main**, the function must not access *any* data objects with dynamic lifetime. All such data objects may be deallocated before the function is called.

The function must return the value returned by **onexit** when you called **onexit** with the pointer to the function. **onexit** uses the value the function returns as the address of the function earlier stored in the static data object. In this way, all functions called by **exit** are called in the reverse order that you requested.

RETURNS

onexit returns a function pointer if the argument is valid. Otherwise it returns a null pointer.

EXAMPLE

To remove your temporary files before program termination:

```
static onexit_t nextfunc;

static char *temp_file[NFILES];

static onexit_t cleanup()
     {
     int i;

     for (i = 0; i < NFILES; ++i)
          remove(temp_file[i]);
     return (nextfunc);
     }
     .....
     nextfunc = onexit(&cleanup);
               /* register cleanup function */
```

SEE ALSO

exit

NAME

rand – generate pseudo random number

SYNOPSIS

```
#include <stdlib.h>
int rand(void);
```

FUNCTION

rand computes successive pseudo random integers in the range [0, 32767]. It uses the following linear multiplicative algorithm, which repeats every 4,294,987,296 times you call it:

```
static unsigned long next = 1;

int rand()
     {
     next = next * 1103515245 + 12345;
     return ((unsigned int)(next / 65536) % 32768);
     }
```

RETURNS

rand returns a pseudo random integer.

EXAMPLE

To flip a coin:

```
printf("%s\n", rand() < 16384
      ? "HEADS" : "TAILS");
```

SEE ALSO

srand

realloc

NAME

realloc – reallocate storage

SYNOPSIS

```
#include <stdlib.h>
void *realloc(void *ptr, size_t nbytes);
```

FUNCTION

realloc alters the amount of storage allocated for a data object earlier allocated by a call to **calloc**, **malloc**, or **realloc**. The data object is at **ptr*. The size in bytes of the new data object is *nbytes*. Storage is not modified within the data object up to the lesser of the old and new sizes. The allocated storage is aligned on a storage boundary acceptable for any data object type.

RETURNS

realloc returns a pointer to the reallocated storage, if there is enough storage available for the new data object. The new pointer may differ from *ptr* even if *nbytes* is smaller than the original object. Otherwise it returns a null pointer, and the old data object is left unaltered.

You must not make use of *ptr* after you call **realloc**.

EXAMPLE

To trim an array of doubles:

```
pd = realloc(pd, n * sizeof (*pd));
```

SEE ALSO

calloc, **free**, **malloc**

NOTES

You must call **realloc** only with a pointer value returned earlier by **calloc**, **malloc**, or **realloc**.

NAME

srand – seed pseudo random number generator

SYNOPSIS

```
#include <stdlib.h>
void srand(unsigned int nseed);
```

FUNCTION

srand uses *nseed* as a seed for the sequence of pseudo random numbers that are returned on calls to **rand**. The same seed value will generate the same sequence of pseudo random numbers. At program startup, the seed value is 1.

RETURNS

Nothing.

EXAMPLE

To repeat a sequence of random numbers:

```
srand(103);
pass1();
srand(103);
pass2();
```

SEE ALSO

rand

strtod

NAME

strtod - convert buffer to floating with error checking

SYNOPSIS

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

FUNCTION

strtod converts the string at *nptr* to a value of type *double*. It interprets the string as the text representation of a real number. The string may have leading whitespace, an optionally signed sequence of decimal digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e**, followed by an optionally signed sequence of decimal digits. The text representation ends just before the first character that does not satisfy this format.

A range error may occur.

If *endptr* is not a null pointer, **strtod** stores the pointer to the first character that is not converted in the data object pointed at by *endptr*. If no characters are converted, the pointer stored is the value *nptr*, even if leading whitespace is present.

RETURNS

strtod returns the converted *double* value. It returns zero if no characters are converted. If a range error occurs, it sets **errno** to **ERANGE** and returns **HUGE_VAL** for values positive and too large to represent. If the value is negative and too large to represent, **strtod** returns **-HUGE_VAL**.

EXAMPLE

To read a string and convert it as a floating constant:

```
fgets(buf, sizeof buf, stdin);
errno = 0;
d = strtod(s, se);
if (errno || *se != '\n')
     printf("bad number\n");
```

SEE ALSO

atof, **atoi**, **atol**, **strtol**

NAME

strtol – convert buffer to long integer with error checking

SYNOPSIS

```
#include <stdlib.h>
long strtol(const char *nptr, char **endptr, int base);
```

FUNCTION

strtol converts the string at *nptr* to a value of type *long*. It interprets the string as the text representation of an integer. The string may have leading whitespace followed by an optionally signed sequence of decimal digits. The text representation ends just before the first character that does not satisfy this format.

A range error may occur.

If *endptr* is not a null pointer, **strtol** stores the pointer to the first character that is not converted in the data object pointed at by *endptr*. If no characters are converted, the pointer stored is the value *nptr*, even if leading whitespace is present.

When *base* is between 2 and 36, it becomes the base for conversion. Leading zeros after the optional sign are ignored, and a leading **0x** or **0X** is ignored if *base* is 16.

If *base* is zero, it is taken as a directive to adapt the base to the input string using the same rules as for integer constants. A leading zero after the optional sign indicates octal conversion. A leading **0x** or **0X** after the optional sign indicates hexadecimal conversion. Otherwise, **strtol** uses decimal conversion.

RETURNS

strtol returns the converted *long* value. It returns zero if no characters are converted. If a range error occurs, it sets **errno** to **ERANGE** and returns **LONG_MAX** for values positive and too large to represent. If the value is negative and too large to represent, **strtol** returns **LONG_MIN**.

EXAMPLE

To read a string and convert it to a *long* constant:

```
fgets(buf, sizeof buf, stdin);
errno = 0;
lo = strtol(s, se, 16);
if (errno || *se != '\n')
     printf("bad number\n");
```

SEE ALSO

atof, **atoi**, **atol**, **strtod**

NAME

system – execute a command

SYNOPSIS

```
#include <stdlib.h>
int system(const char *cmd);
```

FUNCTION

system interprets the string at *cmd* as a command line. It invokes the executable file or builtin command you specify as if you typed the string as a text line to the command interpreter.

Under VM/CMS, **system** uses the SVC202 system service. If your program is executing in the user area, you must not call any CMS command or program that also runs in the user area. The limitations of SVC202 and all CMS commands that run in the user area are described in the *VM System Product CMS User's Guide* (SC19-6210).

You write a command line as an executable file name, in either uppercase or lowercase, followed by any parameters traditionally in uppercase. For example:

```
system ("CC MYPROG (LIST DEFINE(BIG)");
```

Under MVS, **system** issues a LINK SVC system call. MVS searches for the program you specify in the same way as if you called the program from a JCL batch stream, as described in the *MVS JCL Reference Manual* (GC28-0646). You write command options as if you were running the program from a JCL batch stream. For example:

```
system ("PGM=CC,PARM='LIST,DEFINE(BIG)'");
```

Write single quotation marks around the command options if the command options contain spaces or other special characters. Write a single quotation mark that is part of a command option as two consecutive single quotation marks.

RETURNS

system returns a nonzero value if it cannot invoke *cmd* or if the program terminates unsuccessfully.

If *cmd* is a null pointer, **system** returns a value of zero to indicate that it is capable of invoking programs. On systems that cannot invoke programs, **system** returns a nonzero value in this case.

EXAMPLE

To invoke commands from the standard input:

```
while (fgets(buf, sizeof buf, stdin))
     if (buf[0] != '!')
          break;
     else if (system(&buf[1]))
          printf("failed\n");
     else
          printf("done\n");
```

NAME

string.h – header file for string functions

SYNOPSIS

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

FUNCTION

The header file **<string.h>** declares functions useful for manipulating "strings" and "buffers." A string is a data object of type *array of char* whose contents are defined up to and including an array element containing the null character **\0**. A buffer is a data object of type *array of char* whose size is specified by an integer value. If the size value is passed in the argument *n*, the data object is an "*n* character buffer."

Some functions manipulate "bounded strings." The contents of a bounded string are either terminated by a null character, like a string, or specified by an array size, like a buffer, whichever is smaller.

A string or buffer is designated by the value of a pointer to its first, or lowest addressed, element. Such a pointer is declared either as type *pointer to char* or *pointer to void*.

When a function compares two strings or buffers for "lexical order," it compares corresponding elements of the two *array of char* data objects, starting with the first elements. If all elements compare equal, or if there are no elements, the two strings or buffers compare equal. Otherwise, the result of the comparison is the result of comparing the first two *char* elements that are not equal. For example, **""** equals **""**, **"abc"** is less than **"abd"**, and **"abcd"** is greater than **"abc"**.

<string.h> declares the following functions:

memchr – scan buffer for character.

memcmp – compare two buffers for lexical order.

memcpy – copy one buffer to another.

memset – propagate fill character throughout buffer.

strcat – concatenate strings.

strchr – scan string for character.

strcmp – compare two strings for lexical order.

strcpy – copy one string to another.

strcspn – find the end of a span of characters not in set.

strlen – find length of a string.

strncat – concatenate bounded strings.

strncmp – compare two bounded strings for lexical order.

strncpy – copy one bounded string to another.

strpbrk – scan string for any character in set.

strrchr – scan string for last occurrence of character.

strspn – find the end of a span of characters in a set.

strtok – get token from string.

Make sure that no functions access elements outside the storage you have reserved to hold the strings or buffers that the functions manipulate. No checking is performed by the functions.

NAME

memchr - scan buffer for character

SYNOPSIS

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

FUNCTION

memchr scans for the first occurrence of the character *c* in an *n* character buffer starting at *s*. Before the scan is made, *s* is type cast to *pointer to char* and *c* is type cast to *char*.

RETURNS

memchr returns a pointer to the character with lowest address that compares equal to *c*. It returns a null pointer if no character compares equal.

EXAMPLE

To remove all spaces from a buffer:

```
while (s = memchr(buf, ' ', n))
    memcpy(s, s + 1, &buf[--n] - s);
```

SEE ALSO

strchr, **strcspn**, **strpbrk**, **strrchr**, **strspn**

memcmp

NAME

memcmp - compare two buffers for lexical order

SYNOPSIS

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

FUNCTION

memcmp compares two buffers, character by character, for lexical order. The definition of lexical order is described with the header file **<string.h>**. The first buffer starts at *s1*. The second buffer starts at *s2*. Both buffers are *n* characters long. *s1* and *s2* are type cast to *pointer to char* before the comparison.

RETURNS

memcmp returns an integer greater than, equal to, or less than zero, according to whether *s1* is lexically greater than, equal to, or less than *s2*.

EXAMPLE

To look for a prefix match:

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

char *s[] = {"quit",
     "replace",
     "insert",
     "exit",
     NULL};
int cmd_type;

main()
     {
     char *c, *find_prefix();
     int i;

     for (i = 0; *s[i]; ++i)
          if ((c = find_prefix(s[i], strlen(s[i]))) != NULL)
               printf("prefix found\n");
          else
               printf("no prefix found\n");
     }
```

```
char *find_prefix(buf, n)
     char *buf;
     size_t n;
     {
     char **ps;
     extern int cmd_type;
     static char *prefixes[] = {
          "\6insert\1",
          "\6delete\2",
          "\7replace\3",
          "\4quit\7",
          NULL};

     for (ps = &prefixes[0]; *ps; ++ps)
          if ((size_t) **ps <= n && !memcmp(&(*ps)[1],    \
               buf, (size_t) **ps))
               {
               cmd_type = (*ps)[(int) **ps + 1];
               return (&buf[(size_t) **ps]);
               }
     return (NULL);
     }
```

SEE ALSO

strcmp, strncmp

memcpy

NAME

memcpy - copy one buffer to another

SYNOPSIS

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

FUNCTION

memcpy copies the *n* characters starting at location *s2* to the buffer starting at *s1*. *s1* and *s2* are type cast to *pointer to char* before the copy occurs. Characters are copied in unspecified order.

RETURNS

memcpy returns *s1*.

EXAMPLE

To remove all spaces from a buffer:

```
while (s = memchr(buf, ' ', n))
    memcpy(s, s + 1, &buf[--n] - s);
```

SEE ALSO

strcpy, **strncpy**

NOTES

If the destination string overlaps the source, and

```
(char *)s2 < (char *)s1)
```

the resulting string at *s1* may not compare equal to the original string at *s2*.

NAME

memset – propagate fill character throughout buffer

SYNOPSIS

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

FUNCTION

memset copies *c* into each element of the *n* character buffer starting at *s*. *s* is type cast to *pointer to unsigned char* before the copy operation is performed.

RETURNS

memset returns *s*.

EXAMPLE

To write **BUFSIZ** null characters to a file:

```
fwrite(memset(buf, '\0', BUFSIZ), BUFSIZ, 1, pf);
```

SEE ALSO

memcpy

NAME

strcat – concatenate strings

SYNOPSIS

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

FUNCTION

strcat copies the string at *s2* to the end of the string at *s1*. The first character of *s2* replaces the null character at the end of *s1*. The null character at the end of *s2* is the last character copied.

RETURNS

strcat returns *s1*.

EXAMPLE

To place the strings "**first string**" and "**second string**" in **buf**:

```
strcpy(buf, "first string");
strcat(buf, " second string");
```

SEE ALSO

strncat

NAME

strchr - scan string for character

SYNOPSIS

```
#include <string.h>
char *strchr(const char *s, int c);
```

FUNCTION

strchr scans for the first occurrence of the character *c* in the string starting at *s*. *c* is type cast to *char* before the scan.

RETURNS

strchr returns a pointer to the character with lowest address that compares equal to *c*. It returns a null pointer if no character compares equal.

EXAMPLE

To remove all spaces from a string:

```
while (s = strchr(str, ' '))
    strcpy(s, s + 1);
```

SEE ALSO

memchr, **strcspn**, **strpbrk**, **strrchr**, **strspn**

strcmp

NAME

strcmp - compare two strings for lexical order

SYNOPSIS

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

FUNCTION

strcmp compares two strings, character by character, for lexical order. The definition of lexical order is described with the header file **<string.h>**. The first string starts at *s1*. The second string starts at *s2*. The comparison includes the terminating null characters.

RETURNS

strcmp returns an integer greater than, equal to, or less than zero, according to whether *s1* is lexically greater than, equal to, or less than *s2*.

EXAMPLE

To look for a string match:

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

main()
     {
     int i;
     static char *s[] =
          {
          "insert",
          "delete",
          "insert",
          "forget",
          NULL
          };

     for (i = 0; *s[i]; ++i)
          if (match_string(s[i]))
               printf("match made\n");
          else
               printf("no match made\n");
     }
```

```
int match_string(str)
     char *str;
     {
     char **ps;
     static char *strings[] =
          {
          "\1insert",
          "\2delete",
          "\3replace",
          "\7quit",
          NULL
          };

     for (ps = &strings[0]; *ps; ++ps)
          if (!strcmp(&(*ps)[1], str))
               return ((size_t) **ps);
     return (0);
     }
```

SEE ALSO

memcmp, strncmp

strcpy

NAME

strcpy - copy one string to another

SYNOPSIS

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

FUNCTION

strcpy copies the string starting at *s2* to the string starting at *s1*. Characters are copied in unspecified order. The terminating null character is copied.

RETURNS

strcpy returns *s1*.

EXAMPLE

To remove all spaces from a string:

```
while (s = strchr(str, ' '))
    strcpy(s, s + 1);
```

SEE ALSO

memcpy, **strncpy**

NOTES

If the destination string overlaps the source, and

```
(char *)s2 < (char *)s1)
```

the resulting string at *s1* may not compare equal to the original string at *s2*.

NAME

strcspn – find the end of a span of characters in a set

SYNOPSIS

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

FUNCTION

strcspn scans the string starting at *s1* for the first occurrence of a character in the string starting at *s2*. It computes a subscript *i* such that

* *s1[i]* is a character in the string starting at *s1*
* *s1[i]* compares equal to some character in the string starting at *s2*, which may be its terminating null character.

RETURNS

strcspn returns the lowest possible value of *i*. *s1[i]* designates the terminating null character if none of the characters in *s1* are in *s2*.

EXAMPLE

To find the start of a decimal constant in a text string:

```
if (!str[i = strcspn(str, "0123456789+-")])
    printf("can't find number\n");
```

SEE ALSO

memchr, **strchr**, **strpbrk**, **strrchr**, **strpn**

strlen

NAME

strlen - find length of a string

SYNOPSIS

```
#include <string.h>
size_t strlen(const char *s);
```

FUNCTION

strlen scans the string starting at *s* to determine the number of characters before the terminating null character.

RETURNS

strlen returns the number of characters in the string before the terminating null character. This number is zero for the null string "".

EXAMPLE

To output a string:

```
fwrite(s, strlen(s), 1, stdout);
```

NAME

strncat - concatenate bounded strings

SYNOPSIS

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

FUNCTION

strncat copies the bounded string at *s2* to the end of the string at *s1*. The first character of *s2* replaces the null character at the end of *s1*. At most *n* characters are copied. If the last character copied is not the null character at the end of *s2*, **strncat** stores a null character immediately after the last character copied.

RETURNS

strncat returns *s1*.

EXAMPLE

To concatenate two strings:

```
char buf[BUFSIZ];

buf[0] = '\0';
strncat(buf, s1, BUFSIZ - 1);
strncat(buf, s2, BUFSIZ - strlen(buf) - 1);
```

SEE ALSO

strcat, **strncpy**

strncmp

NAME

strncmp - compare two bounded strings for lexical order

SYNOPSIS

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

FUNCTION

strncmp compares two bounded strings, character by character, for lexical order. The definition of lexical order is described with the header file **<string.h>**. The first string starts at *s1*. The second string starts at *s2*. At most *n* characters are compared. The comparison includes the terminating null characters.

RETURNS

strncmp returns an integer greater than, equal to, or less than zero, according to whether *s1* is lexically greater than, equal to, or less than *s2*.

EXAMPLE

To look for a string match:

```
int match_string(buf, n)
     char *buf;
     size_t n;
     {
     char **ps;
     static char *strings[] = {
          "\1insert",
          "\2delete",
          "\3replace",
          "\7quit",
          NULL};

     for (ps = &strings[0], *ps; ++ps)
          if (!strncmp(&(*ps)[1], buf, n))
               return ((*ps)[0]);
     return (0);
     }
```

SEE ALSO

memcmp, **strcmp**

NAME

strncpy – copy one bounded string to another

SYNOPSIS

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

FUNCTION

strncpy copies the bounded string starting at *s2* to the string starting at *s1*. Characters are copied in unspecified order. If the terminating null character is copied before *n* characters have been copied, **strncpy** stores null characters immediately following the last character copied until *n* characters have been copied or stored.

If *n* characters are copied and the terminating null character has not been copied, the resulting bounded string does not have a terminating null character.

RETURNS

strcpy returns *s1*.

EXAMPLE

To remove all spaces from a string:

```
while (s = strchr(str, ' '))
    strncpy(s, s + 1, strlen(s));
```

SEE ALSO

memcpy, **strcpy**

NOTES

If the destination string overlaps the source, and

```
(char *)s2 < (char *)s1)
```

the resulting string at *s1* may not compare equal to the original string at *s2*.

strpbrk

NAME

strpbrk – scan string for any character in set

SYNOPSIS

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

FUNCTION

strpbrk scans the string starting at *s1* for the first occurrence of a character in the string starting at *s2*. It computes a subscript *i* such that

* *s1[i]* is a character in the string starting at *s1*
* *s1[i]* compares equal to some character in the string starting at *s2*, which may be its terminating null character.

RETURNS

strpbrk returns *&s1[i]* for the lowest possible value of *i*, or a null pointer if *s1[i]* designates the terminating null character.

EXAMPLE

To replace all whitespace characters with spaces:

```
while (s = strpbrk(s, "\n\f\r\t\v"))
    *s = ' ';
```

SEE ALSO

memchr, **strchr**, **strcspn**, **strrchr**, **strspn**

NAME

strrchr – scan string for last occurrence of character

SYNOPSIS

```
#include <string.h>
char *strrchr(const char *s, int c);
```

FUNCTION

strrchr scans for the last occurrence of the character *c* in the string starting at *s*. *c* is type cast to *char* before the scan.

RETURNS

strrchr returns a pointer to the character with highest address that compares equal to *c*. It returns a null pointer if no character compares equal.

EXAMPLE

To find the last "word" in a text line:

```
s = strrchr(str, ' ');
if (*s != ' ')
     s = str;
```

SEE ALSO

memchr, **strcspn**, **strpbrk**, **strrchr**, **strspn**

strspn

NAME

strspn - find the end of a span of characters not in set

SYNOPSIS

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

FUNCTION

strspn scans the string starting at *s1* for the first occurrence of a character not in the string starting at *s2*. It computes a subscript *i* such that

* *s1[i]* is a character in the string starting at *s1*
* *s1[i]* compares equal to no character in the string starting at *s2*, except possibly its terminating null character.

RETURNS

strspn returns the lowest possible value of *i*. *s1[i]* designates the terminating null character if all of the characters in *s1* are in *s2*.

EXAMPLE

To check a string for characters other than decimal digits:

```
if (str[strspn(str, "0123456789")])
    printf("invalid number\n");
```

SEE ALSO

memchr, **strcspn**, **strchr**, **strpbrk**, **strrchr**

NAME

strtok - get token from string

SYNOPSIS

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

FUNCTION

strtok helps you group a string into "tokens." A token is a span of characters containing no separator characters from the string *s2*.

If *s1* is not a null pointer, **strtok** scans the string starting at *s1*. Otherwise, it scans the string starting at the pointer value it stored, in an internal static data object, from the last time your program called **strtok**.

strtok scans the string for a character not in the string *s2*. If it encounters the terminating null character before it finds a character not in the string *s2*, there are no more tokens. The pointer value it stores designates the terminating null character. It returns a null pointer.

Otherwise, the token starts with the character it found. **strtok** scans from this point for a character in the string *s2*. If it encounters the terminating null character before it finds a character in the string *s2*, there are no more tokens after this one. The pointer value it stores designates the terminating null character. It returns a pointer to the start of the token.

Otherwise, the token ends with the character it found. **strtok** stores a null character in place of the character it found. The pointer value it stores designates the next character after the character it found. It returns a pointer to the start of the token.

You may specify a different string *s2* each time you call **strtok**. Do not store into the string that **strtok** is scanning.

RETURNS

strtok returns a pointer to the start of the next token, if there is one. Otherwise it returns a null pointer.

EXAMPLE

To parse a text line into "words:"

```
for (s = str; pw = strtok(s, " ,.\":;"); s = NULL)
     spell_check(pw);
```

time.h

NAME

time.h – header file for timekeeping functions

SYNOPSIS

```
#include <time.h>
```

FUNCTION

The header file **<time.h>** declares functions used for timekeeping.

<time.h> defines the following types:

clock_t – an arithmetic type which holds "clock ticks," used to measure elapsed time during program execution. There are **CLK_TCK** clock ticks per second.

time_t – a scalar type which holds "date and time," used to determine a given moment in time. You may not perform arithmetic on values of type **time_t**. Use the function **difftime** to determine the difference between two times.

struct tm – a *struct tag* that designates a type whose structure members represent different components of a date and time, broken down into conventional divisions. The following structure members are part of its content:

```
int tm_sec; /* seconds after the minute [0, 59] */
int tm_min; /* minutes after the hour [0, 59] */
int tm_hour; /* hours since midnight [0, 23] */
int tm_mday; /* day of the month [1, 31] */
int tm_mon; /* month of the year [0, 11] */
int tm_year; /* years since 1900 */
int tm_wday; /* days since Sunday [0, 6] */
int tm_yday; /* day of the year [0, 365] */
int tm_isdst; /* daylight savings time (-1 = status
     unknown; 0 = not in effect; 1 = in effect) */
```

The structure members are not necessarily in this order, and there may be additional structure members.

<time.h> defines the following macro:

CLK_TCK – the number of clock ticks per second. It expands to a constant expression of type **clock_t**.

<time.h> declares the following functions:

asctime – convert time structure to string.

clock – get clock ticks.

ctime – convert time to string.

difftime – calculate difference between times.

gmtime – convert time to Greenwich Mean Time.

localtime – convert time to local time.

time – get time.

NAME

asctime – convert time structure to string

SYNOPSIS

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

FUNCTION

asctime converts the time structure pointed at by *timeptr* to a 26 character string. The length of each field is always the same, regardless of the value it represents. The string has the form:

```
Tue Aug 01 09:00:00 1978\n\0
```

The fields are, in order:

* The first three characters of the day of the week, followed by a space, as in: **Sun Mon Tue Wed Thu Fri Sat**.
* The first three characters of the month, followed by a space, as in: **Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec**.
* The two digit day of the month, followed by a space, as in: **01** through **31**.
* The two digit hour of the day, followed by a colon, as in: **00** through **23**.
* The two digit minutes past the hour, followed by a colon, as in: **00** through **59**.
* The two digit seconds past the minute, followed by a space, as in: **00** through **59**.
* The four digit year, followed by a newline and a null character.

The string is stored in an internal data object.

RETURNS

asctime returns a pointer to the date string.

EXAMPLE

To print the date and time:

```
time_t lt;

time(&lt);
printf("%s", asctime(localtime(&lt)));
```

SEE ALSO

clock, **ctime**, **difftime**, **gmtime**, **localtime**, **time**

NOTES

Make full use of the stored value before you call **asctime** or **ctime** again, since it will be altered on each call.

clock

NAME

clock - get clock ticks

SYNOPSIS

```
#include <time.h>
clock_t clock();
```

FUNCTION

clock gets the elapsed time during program execution, as measured in "clock ticks", which may be unique to each system. You subtract the values returned on two calls to **clock** to determine how much time your program has consumed between calls. You convert this difference to a time in seconds by dividing by **CLK_TCK**.

RETURNS

clock returns the cumulative time that your program has executed, measured in clock ticks from an unspecified origin. If you move your program to a system that has no clock, **clock** returns the value -1. Under VM/CMS, MVS, and MVS/XA **clock** is not implemented and always returns failure.

EXAMPLE

To compute and print elapsed time:

```
t = clock();
do_timed_process();
t = clock() - t;
printf("elapsed time %.2f seconds\n", ((double) \
                t / CLK_TCK));
```

SEE ALSO

asctime, **ctime**, **gmtime**, **localtime**

NAME

ctime – convert time to string

SYNOPSIS

```
#include <time.h>
char *ctime(char time_t *timer);
```

FUNCTION

ctime converts the time pointed at by *timer* to local time, represented as a time structure of type **struct tm**. It then converts the time structure to a 26 character string. The format of the string is exactly the same as described in **asctime**. **ctime(***timer***)** is entirely equivalent to **asctime(localtime(***timer***))**.

The string is stored in an internal data object.

RETURNS

ctime returns a pointer to the date string.

EXAMPLE

To print the date and time:

```
time_t lt;

time(&lt);
printf("%s\n", ctime(&lt));
```

SEE ALSO

asctime, **clock**, **difftime**, **gmtime**, **localtime**, **time**

NOTES

Make full use of the stored value before you call **asctime** or **ctime** again, since it will be altered on each call.

difftime

NAME

difftime – find difference between times

SYNOPSIS

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

FUNCTION

difftime computes the difference between *time2* and *time1* in seconds.

RETURNS

difftime returns the difference in seconds between the two times.

EXAMPLE

To test if **otime** was at least five minutes ago:

```
if (5 * 60.0 <= difftime(time(NULL), otime))
    retry();
```

SEE ALSO

asctime, **clock**, **ctime**, **gmtime**, **localtime**, **time**

NAME

gmtime – convert time to Greenwich Mean Time

SYNOPSIS

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

FUNCTION

gmtime converts the time pointed at by *timer* to a time structure whose components represent Greenwich Mean Time (GMT).

The structure is stored in an internal data object.

RETURNS

gmtime returns a pointer to the time structure.

EXAMPLE

To print the date as GMT:

```
time_t lt;

lt = time(NULL);
printf("%s GMT\n", asctime(gmtime(&lt)));
```

SEE ALSO

asctime, **ctime**, **difftime**, **localtime**, **time**

NOTES

Make full use of the stored value before you call **gmtime** or **localtime** again, since it will be altered on each call.

localtime

NAME

localtime - convert time to local time

SYNOPSIS

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

FUNCTION

localtime converts the time pointed at by *timer* to a time structure whose components represent local time.

The structure is stored in an internal data object.

RETURNS

localtime returns a pointer to the time structure.

EXAMPLE

To print the date as local time:

```
time_t lt;

lt = time(NULL);
printf("It is now %s\n", asctime(localtime(&lt)));
```

SEE ALSO

asctime, **ctime**, **difftime**, **gmtime**, **time**

NOTES

Make full use of the stored value before you call **gmtime** or **localtime** again, since it will be altered on each call.

NAME

time - get time

SYNOPSIS

```
#include <time.h>
time_t time(time_t *timer);
```

FUNCTION

time gets the current date and time. If *timer* is not a null pointer, the current date and time are stored in the data object pointed at by *timer*.

RETURNS

time returns the current date and time. It may also store it at *timer*, if *timer* is not a null pointer. If you move your program to a system that cannot determine the current date and time, the value returned by **time** is -1.

EXAMPLE

To print the local date and time:

```
time_t lt;

time(&lt);
printf("It is now %s\n", ctime(&lt));
```

Appendix A: Compile Time Error Messages

There are three sources of compile time errors:

* Errors in your program. You have written an incorrect program, one that violates one or more of the rules for writing C programs. Alter your source file to correct these errors.

* Environmental problems. You have misnamed a file, or failed to give access permissions that the compiler needs to read or write it. Alter the names of the files, or the file names you specify to the compiler, to correct these problems. The compiler may fail to create a temporary intermediate file, or to write it completely, because there is inadequate space or because it cannot create temporary files. Alter the environment to meet the needs of the compiler. Error messages that the **RLINK** program generates are also described here.

* Errors in the compiler. The compiler detects an inconsistent internal state and reports it. Such occurrences are uncommon. These messages contain an exclamation point, to distinguish them from other messages. You are requested to report the occurrence of such errors, along with the inputs that caused the error report, to your service representative.

This appendix documents the error messages in the first category, followed by the error messages in the second category. Those in the third category should occur very rarely or not at all, and are not described. The description of each error message suggests possible sources of the error, and ways to correct it.

Errors in Your Program

"(" required

The compiler expects a left parenthesis where none is present, such as after the keyword **if**.

")" required

The compiler expects a right parenthesis where none is present. You may have written parentheses that do not balance. You may have written something the compiler does not expect.

")" required in "#if" expression

You have written a **#if** or **#elif** directive, and the expression requires a right parenthesis where none is present. You may have written parentheses that do not balance. You may have written something the compiler does not expect inside parentheses.

":" required

The compiler expects a colon where none is present, such as after a case label. You may have written something the compiler does not expect.

":" required after "?" in "#if" expression

You have written a **#if** preprocessor directive containing a **?**, and the compiler does not encounter a **:** where it expects one. You write the conditional operator as **x?y:z**. You may have written parentheses that do not balance.

":" required

The compiler expects a semicolon where none is present, such as after the keyword **break**. You may have written something the compiler does not expect.

"]" required

The compiler expects a right bracket where none is present, such as after a subscript expression. You may have written brackets that do not balance. You may have written something the compiler does not expect.

"{" required

The compiler expects a left brace where none is present. You may have written something the compiler does not expect.

"}" required

The compiler expects a right brace where none is present. You may have written braces that do not balance. You may have written something the compiler does not expect.

"##" permitted only in macro definition

You have written the preprocessor operator **##** in a context other than a macro definition. You may only use this operator within a preprocessor **#define** directive. See Chapter 7, "The Preprocessor," for information on how to use the **##** operator.

argument already declared: <name>

You have written an argument level declaration that declares the same identifier twice in the argument list in the function declaration. The identifier is *name*. You may declare an argument at most once in argument level declarations. You may have misspelled the identifier.

argument cannot have incomplete type

You have written a function prototype and one of the arguments is declared as having type **void**. Or, you have written a function call and one of the arguments is a structure of unknown content. An array of unknown size is converted to a pointer argument, but no other incomplete type is permitted.

arithmetic operand required

You have written an expression containing an operator that requires one or more arithmetic operands. Some operators that require arithmetic operands are multiply **x*y**, divide **x/y**, remainder **x%y**, add **x+y**, and subtract **x-y**. An arithmetic type is either an integer or a floating type. The operand in question may be a pointer type, a structure type, or type *void*. You may have specified an invalid combination of pointer and other type operands for an add or subtract operator.

array size unknown

You are referring to an array of unknown content at a point where the compiler must know the size in bytes of data objects with that array type. You may have omitted the size of an array in a declaration, as in **char a[]**, and specified no data initializer for the array. If the compiler must allocate storage for the data object, or if you write an expression such as **sizeof a**, for instance, the compiler will emit this error message.

bitfield size out of range

You have declared a bitfield and the constant integer expression following the colon does not have a value in the range [0, 32]. Or, the value is zero and you have specified a name for the bitfield.

bitfield size undefined

You have written an expression that contains the **sizeof** operator, and the operand you have associated with the **sizeof** operator is an lvalue that designates a bitfield. You may not specify a bitfield as the operand of the **sizeof** operator.

cannot take address of operand

You have written an expression containing the form **&x**, where **x** is a void expression, an rvalue, an lvalue with some bitfield type, or a data object identifier declared with storage class **register**. You may take the address of a function designator, or of an lvalue that is not of some bitfield type and does not have storage class **register**. If **x** is an identifier that a header file defines as a macro, the compiler may emit this error message.

"case" value already used

You have written a *case* label whose value matches the value of another *case* label within the same *switch* statement. All *case* label values are type cast to the same integer type as the expression in the *switch* statement. The resulting values must all be distinct. You may have written *case* label expressions using different macro identifiers that are defined as expressions with the same value. You may have misplaced braces, or eliminated a *switch* statement without reviewing its controlled compound statement.

const modified

You have written an expression that has an operand of a const type where the compiler requires a modifiable lvalue. The increment and decrement operators, such as **x++**, and the assigning operators, such as **x=y** and **x+=y** all require that **x** be a modifiable lvalue. You may be assigning to a structure member of a const type in a structure that is not of a const type, or you may be assigning to a structure member that is not of a const type in a structure that is of a const type. You may be assigning to an lvalue of the form ***p**, where **p** is a pointer to a const type.

constant integer expression required

You have written an expression that is not a constant integer expression, and the compiler requires a constant integer expression. The compiler requires a constant integer expression as the value of a *case* label, as the size of a bitfield, as the value of an enumeration constant, and as the optional size of an array declarator. You may have written operators or operands that the compiler cannot evaluate to an integer at compile time, such as **x++**, **2.3**, or **&a[5]**. You can write a constant integer expression that contains floating constants and performs arithmetic upon them, but you must also write a type cast operator to convert a floating operator result to an integer type.

directive not allowed within macro arguments

You have written a macro expansion containing an argument list, and a preprocessor directive occurs in your source text before the end of the argument list. Write the macro name and argument list on adjacent lines within one source file, with no intervening preprocessor directives. You may not conditionally skip part of the argument list, as with the **#if** and **#endif** preprocessor directives.

"#endif" required

You have written a **#if**, **#ifdef**, or **#ifndef** preprocessor directive, and the balancing **#endif** is not within the same source file. You must contain a conditional preprocessor directive group completely within one source file.

enumeration constant already declared: <name>

You have written a declaration containing an enum whose content declares an identifier as an enumeration constant, and that identifier has already been declared within the current block. The identifier is *name*. You may have misspelled the identifier.

enumeration constant out of range: <name>

You have specified a value for an enumeration constant that cannot be represented as type *int*. The enumeration constant has the name *name*. For example,

```
enum End_points {
     LOW = INT_MIN, LOW1,
     HIGH1 = INT_MAX - 1, HIGH,
     LAST};              /* ERROR: value is INT_MAX + 1 */
```

If you specify an octal or hexadecimal integer constant with its sign bit set, the compiler cannot represent its value as type *int*. On System/370, you should write **-1** instead of **0xFFFFFFFF**.

expression required

The compiler expects an expression where none is present. You may have written something the compiler does not expect. You may have misspelled a keyword. You may not omit expressions within a comma separated list, as in **f(a,,c)**.

expression too large

You have written a complex expression that generates more than 4,096 bytes of executable code. The compiler cannot process an expression that generates more than 4,096 bytes of executable code. Rewrite the expression as two or more separate expressions.

external names conflict: <name>

You have declared two different identifiers with external linkage, and the external names derived from the two identifiers are identical. One of the identifiers is *name*. Function names are truncated to eight characters, data object names are truncated to seven characters. Both are compressed to one alphabetic case. The compiler detects conflicts in external names within a compilation. The linkage editor or loader detects conflicts among compilations. If a name need not be linked with other compilations, you can alter its declaration to give it internal linkage or no linkage. Otherwise, you must alter one or both names to resolve the conflict.

fewer function call arguments than in prototype

You have written an expression containing a function call, and the function call has fewer actual arguments than the number of arguments specified in the function prototype.

fewer function definition arguments than in prototype

You have written a function definition that specifies fewer arguments than the number of arguments specified in the function prototype.

floating constant out of range

You have written a floating constant whose value is too large or too small to represent.

floating expression out of range

You have written an expression involving two floating operands and the value of the expression is too large or too small to represent.

function call argument incompatible with prototype

You have written an expression containing a function call, and an actual argument in the function call does not agree with the corresponding argument declaration in the function prototype. The actual argument may not be assignment compatible with the prototype argument, or you may have written the wrong number of actual arguments.

function cannot return const or volatile type

You have written a declaration containing a function type attribute, and the type returned by the function is a const or volatile type. The type qualifiers **const** and **volatile** apply only to lvalues. The value returned by a function is an rvalue.

function cannot return type

You have written a declaration containing a function type attribute, and the function has type *function returning array of T1*, or *function returning function returning T2*. Or, you have written a function definition or a function call and the function has type *function returning structure of unknown content*. You cannot call such functions. In the first case, you can declare the function with type *function returning pointer to T1*. In the second case, you can declare the function with type *function returning pointer to function returning T2*. In the third case, you must specify the content of the structure.

function definition argument conflicts with prototype: <name>

You have written a function definition, and an argument declared in the function definition conflicts with the corresponding argument declaration in the function prototype. The name of the argument is *name*. The definition argument may not be the same type as the prototype argument, or you may have written fewer definition arguments than prototype arguments.

function required for function call

You have written an expression containing the form **x(y)**, and **x** is not of type *function returning T* or *pointer to function returning T*. You may have misspelled the name of a function. If you choose a function name that is a keyword or that a header file defines as an rvalue macro, the compiler may emit this error message.

function size undefined

You have written an expression containing the operator **sizeof** and its operand is some function type. The operand must have a data object type.

identifier already declared in block: <name>

You have written a file level declaration for an identifier previously declared within the same block with no linkage. The identifier is *name*. You may not redeclare an identifier with no linkage. If you declare a data object at block level with any storage class except **extern**, it has no linkage. Note that the declaration sequence:

```
static int x;
extern int x;
```

redeclares **x** at file level. At block level, however, the sequence declares conflicting data objects.

identifier or "{" required

You have written a declaration with the keyword **struct**, **union**, or **enum**, and it is followed by neither a tag identifier nor a content in braces. Specify a tag, a content, or both.

identifier required after "." or "->"

You have written an expression containing one of the operators **.** or **->**, and no identifier follows the operator. Write an identifier that matches the name of a structure member in the structure type specified by the left operand. You may have misspelled the identifier.

identifier required after "#define"

You have written a **#define** preprocessor directive that does not begin with an identifier. Both forms of **#define**, for macros with and without arguments, must start with an identifier.

identifier required after "goto"

You have written the keyword **goto** at the start of a statement, and no identifier follows. Write a label in a *goto* statement. You may have misspelled the identifier.

identifier required after "#ifdef"

You have written a **#ifdef** preprocessor directive with no identifier, or with something on the text line after the identifier. You may also write comments or whitespace on the text line, but nothing else.

identifier required after "#ifndef"

You have written a **#ifdef** or **#ifndef** preprocessor directive with no identifier, or with something on the text line after the identifier. You may also write comments or whitespace on the text line, but nothing else.

identifier required after "#undef"

You have written a **#undef** preprocessor directive that does not begin with an identifier, or with something on the text line after the identifier. You may also write comments or whitespace on the text line, but nothing else. It is not an error to **#undef** an identifier that is not currently defined as a macro.

identifier required for function definition argument

You have written a function definition and one of the arguments in the function attribute is not an identifier. Provide identifiers for all arguments in a function definition.

identifier required in prototype argument

You have written a function definition that is also a function prototype and one of the arguments does not contain an identifier. Provide identifiers for all arguments in a function definition.

incompatible assignment

You have written an expression containing an assignment, and the type you are assigning is not assignment compatible with the destination type. You may have written an expression containing the form **x=y**. You may have written a data initializer in a declaration for a data object with dynamic lifetime, as in:

```
register char *p = get_buf();
```

You can often use a type cast operator to make a scalar rvalue assignment compatible with another scalar type modifiable lvalue.

incompatible return expression

You have written a *return* statement within an expression, and the expression type is not assignment compatible with the type returned by the function. Or, the function is a void function. You may not write an expression in a *return* statement with a void function.

incomplete comment in file

You have written a comment that does not end before the end of the source file in which it begins. Write a comment entirely within one source file.

incomplete function body

You have written a function body, and the compiler encounters the end of your source file before the end of the function body. You may have written braces that do not balance. Your source file may have been truncated. The compiler may have skipped source text while recovering from an earlier error.

insufficient stack

The compiler has exhausted the storage it uses for calling environments and data objects with dynamic lifetime. You may have written a complex macro expansion or a complex expression. Alter your program to lessen the demands it makes on the compiler.

integer constant out of range

You have written an integer constant and the value is too large to be represented as type **unsigned long**.

integer constant required after "#line"

You have written a **#line** preprocessor directive that does not begin with an integer constant. Or you have written something after the integer constant other than an optional file name inside double quotation marks. You may also write comments or whitespace on the text line, but nothing else.

integer expression required for "switch" statement

You have written a *switch* statement, and its expression is not of integer type. You may not write a *switch* statement expression of floating or pointer type. You may write a type cast to convert any arithmetic type to an integer type, as in:

```
switch ((int)log10(answer))
```

integer operand required

You have written an expression containing an operator that requires one or more integer operands. Some operators that require integer operands are remainder **x%y**, bitwise OR **x|y**, and bitwise NOT **~x**. The operand in question may be a floating type, a pointer type, a structure type, or type *void*.

invalid base type for bitfield

You have written a structure declaration containing a bitfield declaration, and the type of the declarator is not *int*, *signed int*, or *unsigned int*. You may not declare a bitfield as, for example:

```
struct {
    short sh_field : 3;     /* ERROR: must be int type */
```

invalid binary operator in "#if" expression

You have written a **#if** or **#elif** preprocessor directive, and you have written a binary operator not acceptable within **#if** expressions. You may have written a comma operator **,** or one of the assigning operators, such as **x+=y**. You may have written = instead of ==.

invalid bitfield initializer

You have written a data initializer for a static data object of bitfield type, and the expression is not a constant integer expression.

invalid character: <x>

You have written a character that is not part of the C source character set, and it is not inside a comment, a character constant, or a string constant. The character is *x*. If it is not a printable character, you may not see it in the error message. Remove this character from your source file.

invalid combination of type specifiers

You have written a declaration whose base type contains a combination of type specifiers that the compiler does not recognize, such as **signed float**.

invalid data initializer

You have specified the **RENT** option to **CC** and your program attempts to initialize a data object of storage class **static** or **extern** with the address of a non "const" static data object. A restriction prohibits this type of initialization when you specify the **RENT** option. This restriction may be removed in a future release.

invalid declarator following "("

You have written a declarator containing parentheses, and the compiler does not recognize a declarator inside the parentheses. If you omit the identifier in a function declaration, the compiler may emit this error message when it encounters the parenthesized argument list. If you choose an identifier that is a keyword or that a header file defines as a macro, the compiler may emit this error message. You may have difficulty expressing a complex declarator properly.

invalid declarator following ""*

You have written a declarator containing an asterisk, and the compiler does not recognize a declarator following the asterisk. If you choose an identifier that is a keyword or that a header file defines as a macro, the compiler may emit this error message. You may have difficulty expressing a complex declarator properly.

invalid "#define" argument list

You have written a macro expansion whose **#define** preprocessor directive has a left parenthesis immediately after the identifier, and the argument list signalled by that left parenthesis is not well formed. Or, the number of arguments written in the expansion does not match the number written in the **#define** preprocessor directive. You may not have intended to specify an argument list in the **#define** preprocessor directive. For a macro with no arguments, you must leave whitespace between the identifier and a definition that begins with a left parenthesis. You may have omitted a comma between argument names, or you may have omitted the right parenthesis that ends the argument list. You may have omitted an argument name or repeated one.

invalid file level declaration

You have written a file level declaration, and the compiler cannot match it to any of the formats for declarations. You may have misspelled a storage class or type keyword. You may have written no declarators, as in

```
extern int;
```

You may have misplaced a semicolon, as in

```
int func();          /* declaration complete here */
    {                /* ERROR: unexpected function body */
    .....
    };          /* ERROR ; permitted after function body */
int x[2] = {1, 2}   /* ERROR: ; required here */
int y;              /* error reported on this line */
```

You may have written braces that do not balance in a data initializer or in a function body. If you choose an identifier that is a keyword or that a header file defines as a macro, the compiler may emit this error message.

invalid floating constant

You have written a floating constant that contains digits or letters that match none of the formats for a floating constant. You may have omitted both the decimal fraction and decimal exponent of a *float* constant, as in **35F**. You may have written conflicting suffixes, as in **3.0LF**. You may have omitted a comma or an operator.

invalid floating initializer

You have written a data initializer for a static data object of floating type, and the expression is not a constant expression of arithmetic type. The macro **HUGE_VAL** is an rvalue of type *double*. It may not be used as a static data initializer.

invalid hexadecimal escape sequence

You have written an expression that includes a character constant or a string constant containing an escape sequence beginning with **\x**, and no hexadecimal digits follow. You must write one, two or three hexadecimal digits immediately following the **\x**. See Chapter 2, "Elements of the C Language," for information on hexadecimal value escape sequences.

invalid integer constant

You have written an integer constant that contains digits or letters that match none of the formats for an integer constant. You wrote the integer constant as *n*. You may have written a digit for another base, as in **08**. You may have repeated a suffix, as in **0x400ULU**. You may have omitted a comma or an operator.

invalid integer initializer

You have written a data initializer for a static data object of integer type, and the expression is not a constant integer expression.

invalid macro argument list

You have expanded a macro with arguments and the argument list is not well formed. You may have written parentheses that do not balance in the argument list. You may have written a preprocessor directive within the argument list. You may have written an argument list not entirely contained within one file. Or, you may have written an argument list that spans text lines totalling more than 511 characters.

invalid operand for '.' or '->'

You have written an expression containing the form **x.y** or **x->y** and **x** does not have the proper type. For **x.y**, **x** must be a structure type. For **x->y**, **x** must be a pointer to a structure type.

invalid operand for "defined" in "#if" expression

You have written a **#if** or **#elif** preprocessor directive containing the special operator **defined**, and the operator is not followed by either an identifier or an identifier in parentheses, such as **defined x** or **defined (x)**. The identifier **defined** is not otherwise reserved, so you may have defined a macro with this name earlier and referred to the macro within the expression.

invalid operand for "#line"

You have written a **#line** preprocessor directive with something after the integer constant other than an optional file name inside double quotation marks. You may also write comments or whitespace on the text line, but nothing else.

invalid operand in "#if" expression

You have written a **#if** or **#elif** preprocessor directive, and you have written an operand not acceptable within **#if** expressions. You may only write integer constants, character constants, and identifiers as simple operands within **#if** expressions. Do not write floating constants or string constants. You may write comments or whitespace after the expression on the text line, but nothing else. You may have written parentheses that do not balance within the expression. You may have written an operator not acceptable within a **#if** preprocessor directive.

invalid operand types for "+="

You have written an expression containing the form **x+=y**, where **x** is a data object pointer type and **y** is not an integer type. You may not add a floating type to a pointer.

invalid operand types for "-="

You have written an expression containing the form **x-=y**, where **x** is a data object pointer type and **y** is not an integer type. You may not subtract a floating type from a pointer or subtract a pointer from a pointer.

invalid operand types for comparison

You have written an expression containing one of the operators **<**, **<=**, **>**, **>=**, **==**, or **!=**, and you are comparing a pointer operand to an operand whose type is not acceptable. You cannot compare a floating type to a pointer type, for example, or pointers of different data object type. You may type cast two data object pointers to the common type *pointer to char* and compare them, if they point within the same data object.

invalid operand types for conditional

You have written an expression containing the form **x?y:z**, and the types of **y** and **z** match no valid combination. You may not combine pointer and floating types, for instance, or data object pointers of different types. You may sometimes use a type cast to convert one operand to an acceptable type.

invalid pointer initializer

You have written a data initializer for a static data object of pointer type, and the expression is not a constant pointer expression. You may have taken the address of a data object with dynamic lifetime. You may have specified an expression of pointer type that is not assignment compatible with the pointer type of the data object. You can write a type cast to convert integer types and many pointer types to a given pointer type. For example:

```
int a[10];
char *fill_p = a;
          /* ERROR: expression is "pointer to int" */
```

```
char *fill_q = (char *)a;     /* valid */
```

invalid "#pragma linkage"

You have written a **#pragma** preprocessor directive with a **linkage** option, and the compiler cannot match the format:

#pragma linkage(*name***, OS)**

You may have written **OS** with lowercase letters. You may have omitted a parenthesis or comma.

invalid statement

You have written a function body, and the compiler cannot match the next statement to any of the formats for statements. You may have misspelled a statement keyword. You may have written an *expression* statement that the compiler cannot match to any of its formats for expressions. You may have written braces that do not balance.

invalid storage class

You have written a declaration containing a storage class keyword that is not valid for that declaration context. You may have written a **typedef** storage class in an argument level declaration. Only the storage class **register** is permissible in an argument level declaration, or within a function prototype argument list. You may have altered a declaration with storage class **register** or **auto** from a block level declaration to a file level declaration.

invalid type cast or operand type

You have written an expression containing the form **(***type name***)x**, and *type name* is not acceptable in a type cast. Or the type of **x** may not be type cast to the specified type. You may only write type casts of scalar type. You may not type cast a function pointer to a data object pointer, for instance, or a data object pointer to a function pointer. If both types are scalar types, replace **x** with **(int)x**, yielding a double type cast that is acceptable. This expression will not necessarily be portable to architectures other than System/370, however.

invalid type for prototype argument

You have written a function prototype, and an argument has an incomplete type. You write a function prototype with no arguments as

```
extern int f(void);
```

You cannot otherwise write the type name **void** as a function prototype argument. You may have written a base type that is a structure of unknown content. You may have omitted a type attribute.

invalid unary operator in "#if" expression

You have written a **#if** or **#elif** preprocessor directive, and you have written a unary operator not acceptable within **#if** expressions. You may use only the operators **defined**, +, –, **!**, and ~. You may not write a type cast or the **sizeof** operator.

invalid use of type definition: <name>

You have written an expression containing an identifier that is declared as a type definition. The identifier is *name*. You may have misspelled the identifier name. You may have omitted a block level declaration.

misplaced "break" statement

You have written a *break* statement that is not contained within a *do/while*, *for*, *while*, or *switch* statement. You may have misplaced braces, or eliminated one of these statements without reviewing its controlled statement.

misplaced "case" label

You have written a *case* label that is not contained within a *switch* statement. You may have misplaced braces, or eliminated a *switch* statement without reviewing its controlled statement.

misplaced "continue" statement

You have written a *continue* statement that is not contained within a *do/while*, *for*, or *while* statement. You may have misplaced braces, or eliminated one of these statements without reviewing its controlled statement.

misplaced "default" label

You have written a *default* label that is not contained within a *switch* statement. Or, you have already written a *default* label within the same *switch* statement. You may have misplaced braces, or eliminated a *switch* statement without reviewing its controlled compound statement.

misplaced "#elif"

You have written a **#elif** preprocessor directive, and there is no unbalanced **#if**, **#ifdef**, or **#ifndef** preceding it.

misplaced "#else"

You have written a **#else** preprocessor directive and there is no unbalanced **#if**, **ifdef**, or **#ifndef** preceding it.

misplaced "#endif"

You have written a **#endif** preprocessor directive, and there is no unbalanced **#if**, **#ifdef**, or **#ifndef** preceding it.

modifiable lvalue operand required

You have written an expression containing an assigning operator, and the left operand is not a modifiable lvalue. Or, the operator is an increment or a decrement operator, and the operand is not a modifiable lvalue. A modifiable lvalue is any data object designator that does not have a const type or an array type. The left operand may not be an incomplete type, such as a structure of unknown content. It may not be a function designator. It may not be an rvalue. Several operators always produce rvalues, even when their operands are lvalues. These include the increment and decrement operators, such as **++x**, the comma operator **x,y**, and the conditional operator **x?y:z**. You may not use any of these as a modifiable lvalue.

more function call arguments than in prototype

You have written an expression containing a function call, and you have specified more arguments on the function call than you specified in the function prototype.

more function definition arguments than in prototype

You have written a function definition, and you have specified more arguments in the function definition than you specified in the function prototype.

OS linkage not allowed for C function

You have specified OS linkage for a C function. Do not write C functions with OS linkage.

pointer or array operand required

You have written an expression containing the form ***x**, and **x** is not a pointer or array type. Or, you have written an expression containing the form **x[y]**, and **x+y** is not a pointer type. If **x** is an array type, it will be converted to an rvalue of pointer type when you add **y** to it. If you intend to address absolute locations in memory, type cast the integer expression that specifies the absolute address, as in **(int *)0x40**. Note that this practice is *extremely* nonportable.

"#pragma linkage" conflicts with declaration: <name>

You have written a **#pragma** preprocessing directive that declares *name* to have special linkage, and *name* has already been declared and used without special linkage. Write the **#pragma**, and any supplemental declaration, before you write an expression that calls the function.

prototype argument already declared: <name>

You have written a declaration containing a function prototype, and you have declared two arguments with the same identifier. If you choose to write names for the arguments, all names must be distinct within the function prototype. If you are not defining the

function with this declarator, you need not write names for the arguments.

redeclaration conflicts at block level: <name>

You have written a block level declaration with storage class **extern** for an identifier previously declared with external or internal linkage, and the declarations do not have the same type. You may have misspelled the identifier.

redeclaration conflicts at file level: <name>

You have written a file level declaration for an identifier previously declared, and the declarations have conflicting storage classes or do not have the same type. You may not use the storage class **static** in a redeclaration of an identifier with external linkage. You may have misspelled the identifier.

redefined data object: <name>

You have written a data initializer for a data object, and the data object is declared earlier with a data initializer in the same compilation. The name of the data object is *name*. You may write at most one data initializer for a data object, among all the compilations that make up your program.

redefined function: <name>

You have provided a second defining instance of a function within the same compilation. You may have misspelled one of the function names. You may have tried to declare the function arguments for a prototype using argument level declarations, as in:

```
extern int f(double a, double b)  /* valid prototype */
    {.....}                       /* with definition */
    .....
extern int f(a, b)  /* invalid prototype */
    double a, b;    /* ERROR: signals start of
                       a definition */
extern int x;       /* ERROR: function body now required */
```

Remove one of the defining instances.

redefined label: <name>

You have written a function body containing a plain label whose identifier appears in an earlier plain label in the same function body. The identifier is *name*. Labels do not have block scope. All labels within a function body must be distinct.

redefined macro: <name>

You have written a **#define** preprocessor directive that defines an identifier already defined as a macro. The identifier is *name*. You may redefine a macro only if the new expansion is spelled exactly the same as the previous expansion. Otherwise, you must write a **#undef** preprocessor directive to remove the definition before you can use the same identifier in a **#define** preprocessor

directive.

redefined tag: <name>

You have written a structure type specifier containing both a tag and a content, and the tag has already been declared in the same scope. Struct, union, and enum tags share the same name space.

repeated type specifier

You have written a declaration whose base type contains the same type specifier more than once, such as **long long int**. Or, you have written two type specifiers that always conflict, such as two type definitions, or a *struct* and a *union* type specifier. You may combine a type definition or a composite type only with the type qualifiers **const** and **volatile**.

scalar operand required

You have written an expression with an operand that must be of scalar type, and it is not. A scalar type is an integer type, a floating type, or a pointer type. The test expression of an *if* statement, for instance, must be of scalar type. Do not write a structure expression or a void expression where a scalar type is required.

source line too long

You have written a text line longer than 511 characters. If you end a physical text line with a backslash, the compiler treats that line and the one following as a single text line. If you write a macro expansion whose argument list does not end on the current text line, the compiler treats that line and the one following as a single text line. Several physical text lines may thus be combined to form a single text line. The compiler cannot process a text line that exceeds 511 characters.

string initializer too long

You have written a data initializer that uses a string constant to initialize an array data object, and there are more elements in the string constant than in the array, even omitting the terminating null character. The string constant **"abc"**, for instance, can initialize an array with three or more elements. If the array is of unknown size, this string will define its size as four elements. You cannot use this string to initialize an array of fewer than three elements, however.

struct size unknown

You are referring to a struct of unknown content at a point where the compiler must know the size in bytes of data objects with that structure type. If the compiler must allocate storage for the data object, or if you write an expression such as **sizeof str**, for instance, the compiler will emit this error message.

structure member already declared: <name>

You have written a structure declaration, and its content contains two structure member declarations with the same identifier. The identifier is *name*.

too many initializers

You have written a data initializer that contains more data items than are required to initialize the data object. You may have written braces improperly within a complex data initializer. You may have altered the size of an array, or deleted structure members from a struct, without altering the data initializer to match.

type definition already declared: <name>

You have written a declaration with storage class **typedef** that declares an identifier previously declared within the same block. Or, the previous declaration within the same block is a type definition. The identifier is *name*. You may have misspelled the identifier. You may have written another type specifier with the type definition, as in:

```
typedef int I;
unsigned I ui; /* ERROR: can't add type specifiers
                          to I */
```

unbalanced double quotation marks

You have written a string constant that contains a newline character before the closing double quotation mark. You must write a string constant all on one text line. You can write a backslash at the end of a physical text line to merge that and the next physical text line into one text line. You can write a long string constant across multiple physical text lines this way. Or you can write string constants on successive lines, separated only by whitespace. The compiler concatenates adjacent string constants to form one combined string constant.

unbalanced single quotation marks

You have written a character constant that contains a newline character before the closing single quotation mark. You must write a character constant all on one text line.

undeclared operand: <name>

You have written an expression that contains an identifier not currently declared and not used as the name of a function to call. The identifier is *name*. You must declare a data object identifier before you use it in any way within an expression. You must declare a function identifier before you take its address in an expression. You may have misspelled the identifier.

undefined label: <name>

You have written a function body containing a *goto* statement, and you defined no label that matches the label in the *goto* statement. The label is *name*. You may not transfer control to another function body with a *goto* statement.

undefined "static" function: <name>

You have declared an identifier as a function with internal linkage, and you have provided no defining instance within the compilation. The identifier is *name*. If you declare a function with storage class **static**, then you must write a declaration for that function with a function body somewhere within the compilation.

union size unknown

You are referring to a union of unknown content at a point where the compiler must know the size in bytes of data objects with that structure type. If the compiler must allocate storage for the data object, or if you write an expression such as **sizeof un**, for instance, the compiler will emit this error message.

unknown argument: <name>

You have written an argument level declaration that declares an identifier not in the argument list in the function declaration. The identifier is *name*. You may only declare arguments in argument level declarations. You may have misspelled the identifier. If you choose an identifier that a header file defines as a macro with arguments, the compiler may emit this error message.

unknown directive following "#"

You have written a comment preprocessor directive **#** with something other than comments or whitespace on the text line. You may have misspelled the name of a preprocessor directive.

unknown structure member: <name>

You have written an expression containing one of the forms **x.y** or **x->y**, and **y** is not a structure member name contained in the structure designated by the type of **x**. You must specify only structure member names from the designated structure type.

void size undefined

You are referring to a *void* type at a point where the compiler must know the size in bytes of a data object. If the compiler must allocate storage for the declaration, as in the case

```
auto void x;
```

or if you write an expression such as **sizeof (void)**, for instance, the compiler will emit this error message.

"while" required

You have written a *do/while* statement, and the compiler expects the keyword **while** where none is present. You may have written more than one statement after the keyword **do**, without enclosing them in braces. You may have written something the compiler does not expect.

zero operand after "%"

You have written an expression containing the form **x%y**, where **y** is a constant expression with the value zero. Do not divide by zero.

zero operand after "/"

You have written an expression containing the form **x/y**, where **y** is a constant expression with the value zero. Do not divide by zero.

Environmental Problems

"(" required

You have specified one or more command options to **CC**, and no left parenthesis precedes the command options. This error message is applicable only under VM/CMS.

")" required

You have specified a command option to **CC** that requires information enclosed in parentheses, and no right parenthesis follows the information. This error message is applicable only under VM/CMS.

cannot create file: <file>

You have specified an assembly language source file or a listing file that the compiler cannot create for writing. Or the compiler cannot create temporary intermediate files. The file name is *file*. Provide sufficient disk space, and appropriate access permissions, for compiler output files. Alternatively, the **RLINK** program is unable to create an output file. On VM/CMS, **RLINK** creates the output file **RLOBJ TEXT** * on the first writable minidisk. Make sure a writable minidisk is available to **RLINK**. On MVS and MVS/XA, **RLINK** creates the output dataset **dd:SYSPUNCH**. Make sure you have set up **SYSPUNCH** correctly. You may have misspelled the file name.

cannot open file: <file>

You have specified a C source file that the compiler cannot open for reading. Or the compiler cannot open a temporary intermediate file. The file name is *file*. Provide appropriate access permissions for compiler input files. Alternatively, you have specified an object file that the **RLINK** program cannot open for reading. The file name is *file*. On VM/CMS, the file type is **TEXT**. On MVS and MVS/XA, the input dataset name is **dd:SYSIN**. Provide appropriate access permissions for **RLINK** input files. You may have misspelled the file name.

cannot open "#include" file: <file>

You have specified a C source file or a library header file that the compiler cannot open for reading. The file name is constructed from *file*. Provide appropriate access permissions for compiler include files. You may have misspelled the file name.

cannot read file: <file>

The compiler has encountered a read error on an open file. The file name is *file*. It assumes that the system has already retried the read, and that the error is irrecoverable. Compile the code again, preferably on a different storage device.

cannot write file: <file>

The compiler has encountered a write error on an open file. The file name is *file*. It assumes that the system has already retried the write operation, and that the error is irrecoverable. There may be no additional room on the storage device. Compile the code again, preferably on a different storage device.

csect name too long

You have specified the **CSECT(***name***)** option to **CC**, and the length of the CSECT name *name* exceeds the length allowed. If you specify **CSECT(***name***)** in combination with the **RENT** option, *name* must not exceed seven characters in length. Otherwise *name* must not exceed eight characters in length.

file name required

You have specified no file name to **CC**. You must specify the name of the source file you want to compile. Alternatively, you have specified no file name to the **RLINK** program. On VM/CMS, you must specify the name of one or more input files to **RLINK**, the file type of which must be **TEXT**.

insufficient heap

The compiler has exhausted the storage it has available for building its internal data structures. You may have written a large number of macro definitions, a large number of declarations, or a large function body. You must increase the size of the virtual machine on which the compiler is running, or alter your program to lessen the demands it makes on the compiler.

invalid card

You have specified input to the **RLINK** program which contains invalid records within the object module. Make sure the object module you use as input to **RLINK** is valid output from the C compiler.

invalid characters in csect name

You have specified the **CSECT(***name***)** option to **CC**, and *name* contains a character that is not an uppercase letter **ABCDEFGHIJKLMNOPQRSTUVWXYZ**, a lowercase letter **abcdefghijklmnopqrstuvwxyz**, or a decimal digit **0123456789**. The first character must be an uppercase or lowercase letter. Use only uppercase letters, lowercase letters, and decimal digits to form CSECT names.

invalid combination of options

You have specified two or more command options to **CC** that are inconsistent. An example is the combination **LIST** and **NOASM**, which calls for a listing combining C source lines and assembler output, but specifies that the assembler is not to be run.

invalid option

You have specified a command option to **CC** that **CC** cannot recognize. See the summary of **CC** command options in your *C Compiler User's Guide* for more information.

invalid range for sequence number columns

You have specified the command option **SEQ(m,n)** to **CC** and the column numbers are inconsistent. Either the first column *m* is zero, or the second column *n* is less than *m*. Columns are numbered from 1. If you specify a range, you must specify a range of one or more columns.

invalid search modifier

You have specified the command option **SEARCH(***mod***)** or **LSEARCH(***mod***)** to **CC** under VM/CMS, and the search modifier *mod* is not a single letter or an * character.

invalid sequence option

You have specified a command option **SEQ(***m,n***)** to **CC** that does not consist of two column numbers separated by a comma and enclosed in parentheses.

rent requires csect name

You have specified the **RENT** option to **CC** and have not specified the **CSECT(***name***)** option in combination with **RENT**. You must specify the **CSECT(***name***)** option when you specify **RENT**.

too many macro definitions

You have defined more **#define** macro definitions than **CC** will allow. You can avoid this problem by including macro definitions in a header file.

Appendix B: Runtime Error Messages

Runtime diagnostics result from three sources:

* Internal conditions. Your program reports a signal for which default handling is specified.
* Environmental problems. You have misnamed a file, or failed to give access permissions that your program needs to read or write it. Alter the names of the files, or the file names you specify to your program, to correct these problems.
* Errors in the C runtime. The C runtime detects an inconsistent internal state and reports it. Such occurrences are uncommon. These conditions cause an abnormal exit, or ABEND, with a code greater than 99, to distinguish them from normal exits and abnormal exits due to other causes. Report the occurrence of such errors, along with the conditions that caused the error report, to your service representative.

This appendix documents the error messages in the first category, followed by the error messages in the second category. Those in the third category should occur rarely or not at all, but are briefly summarized with their ABEND codes.

Internal Conditions

Default Signal Handling

If you report a signal by calling the function **kill**, default handling is specified for that signal unless you explicitly alter the handling by calling the function **signal**. Default handling is to print the contents of the machine registers, along with the program status word, by using the WTP system service. Your program then takes an abnormal exit with an ABEND code of:

1 – For the abort signal **SIGABRT**.

2 – For the terminal attention interrupt **SIGINT**.

3 – For the program interrupt **SIGTERM**.

ABEND codes 1 through 9 are reserved for signals.

Program Termination

The function **exit** takes a normal exit. A return code of 0 indicates "successful termination." All other values are various forms of "unsuccessful termination." No message is printed.

Environmental Problems

cannot redirect STDIN

You have specified redirection of the standard input stream by writing a command option of the form < *file*, and the file cannot be opened for reading. *file* is the name of the file. You may have misspelled the file name. The C runtime writes an error message to the standard error stream and calls **exit** with an argument value of 1.

cannot redirect STDOUT

You have specified redirection of the standard output stream by writing a command option of the form > *file*, and the file cannot be created for writing. *file* is the name of the file. You may have misspelled the file name. The C runtime writes an error message to the standard error stream and calls **exit** with an argument value of 1.

command line error

You have written a command line that is invalid. You may have redirected a stream more than once. You may have written quotation marks that do not balance. The C runtime writes an error message to the standard error stream and calls **exit** with an argument value of 1.

Unsupported Operating System

You have invoked your program on a system that the C runtime does not recognize. Your program takes an abnormal exit with an ABEND code of:

10 – For an unsupported operating system.

ABEND codes 10 through 19 are reserved for initialization errors.

Not Enough Space

Your program may make heavy demands for storage. On VM/CMS, you can change the size of the virtual machine with the **CP DEFINE STORAGE** command. On MVS and MVS/XA, you can change the amount of storage available to your program by altering the **REGION** job card parameter. On TSO, you can change the amount of storage available to your program by altering the **SIZE** parameter to the **LOGON** command. If your program has insufficient storage, it takes an abnormal exit with an ABEND code of:

20 – For insufficient main stack.

21 – For insufficient communications area in the first 16 Mbytes of virtual memory. This occurs only under MVS/XA.

22 – For insufficient signal stack.

23 – For insufficient storage for the signal attention routine.

24 – For insufficient space for library static data.

ABEND codes 20 through 29 are reserved for current and future insufficient storage errors.

Errors in the C Runtime

The following errors should occur rarely or not at all. If they do, your program takes an abnormal exit with an ABEND code of:

100 – For an open error on any of the three standard streams.

101 – For an invalid return code from the FREEMAIN system service.

102 – For an invalid return code from the DMSFRET system service.

103 – For an invalid return code from the STAX system service.

104 – For an invalid pointer type passed to the FREEMAIN or DMSFRET system service (storage not allocated by GETMAIN or DMSFREE).

105 – For an invalid attempt to perform an exit when not started by **main** or **dlitc**.

106 – For an unsuccessful attempt to free stack space or system static data area.

107 – For a return from exit.

ABEND codes 100 through 199 are reserved for current and future C runtime errors.

Appendix C: Summary of Reserved Identifiers

The following is a list of all identifiers defined by the compiler or by the C library. The compiler defines only keywords. Each identifier defined by the C library is defined or declared in a header file.

Do not declare a function or data object whose name matches *any* of these reserved identifiers. Do not, as a matter of caution, use any of the identifiers listed below in any way within your programs.

All identifiers that begin with an underscore are reserved for use by the compiler. Assume that any such name might be defined as a macro, even if you include no header files in a compilation.

Usage of Reserved Identifiers

The alphabetical listing of all identifiers also shows the header file that defines or declares it and how it is used. No header file is given for keywords. The usage classifications are:

* If an identifier is a "constant integer macro," you may use it in a `#if` expression, or in any static data initializer.
* If an identifier is a "constant arithmetic macro," you may use it in a static data initializer for a data object of arithmetic type.
* If an identifier is a "constant pointer macro," you may use it in a static data initializer for a data object of pointer type.
* If an identifier is an "rvalue macro," you may *not* use it in a static data initializer. You may use it only in an expression that permits an rvalue of the specified type. An rvalue is permitted wherever a void expression is permitted.
* If an identifier is an "lvalue macro," you may *not* use it in a static data initializer. You may use it only in an expression that permits an lvalue of the specified type. An lvalue is permitted wherever an rvalue or a void expression is permitted.
* If an identifier is a "function only," you may use it in a static data initializer. You can use it for a data object of type pointer, you may take its address, and you may use it in a function call. It is declared within the header file as a function, with external linkage.

* If an identifier is a "function or macro," you may *not* use it in a static data initializer, and you may not take its address. You may use it only in a function call. It is declared within the header file as a function, with external linkage. It may also be subsequently defined in the header file as a macro. It will evaluate each of its arguments exactly once, whether it is implemented as a macro or not. You may remove any macro definition with the **#undef** preprocessor directive.

* If an identifier is a "function or unsafe macro," as a function it behaves just as a "function or macro." As a macro, however, it may evaluate one of its arguments more than once.

* If an identifier is a "statement macro," you may use it only where a statement is permitted. You follow the macro call with a semicolon, just like an expression in an *expression* statement.

* If an identifier is a "type definition," you may use it wherever a type specifier is permitted.

* If an identifier is a "struct tag," you may use it wherever a type specifier is permitted. You write the type specifier as **struct** *tag*, where *tag* is the identifier. It is declared within the header file as a struct tag, along with the content that goes with it.

* If an identifier is a "macro reference," it is *not* defined within the header file. Instead, you must choose whether to define it as a macro before including the header file. You can define a macro either with a **#define** preprocessor directive, or on the command line that invokes the C compiler.

* If an identifier is a "function reference," it is *not* defined within the C library. You must provide a definition for this function in one of the compilations that make up your program.

* If an identifier is a **#if** or **#elif** operator, it has special meaning only within an expression in one of those preprocessor directives.

If the word "added" appears before any of these usages, the identifier is *not* in the proposed ANSI Standard. Its usage in the C library is an extension.

If the word "old" appears before any of these usages, the C library offers a function that subsumes or replaces the function declared with that identifier. Use the replacement suggested at the end of the description of the "old" function.

The Reserved Identifiers

The reserved identifiers are:

Header File	*Identifier*	*Usage*
stdio.h	BUFSIZ	constant integer macro
limits.h	CHAR_BIT	constant integer macro
limits.h	CHAR_MAX	constant integer macro
limits.h	CHAR_MIN	constant integer macro
time.h	CLK_TCK	constant arithmetic macro
limits.h	DBL_DIG	constant integer macro
limits.h	DBL_MAX_EXP	constant integer macro
limits.h	DBL_MIN_EXP	constant integer macro
limits.h	DBL_RADIX	constant integer macro
limits.h	DBL_ROUNDS	constant integer macro
math.h	EDOM	constant integer macro
stdio.h	EOF	constant integer macro
math.h	ERANGE	constant integer macro
stdio.h	FILE	type definition
limits.h	FLT_DIG	constant integer macro
limits.h	FLT_MAX_EXP	constant integer macro
limits.h	FLT_MIN_EXP	constant integer macro
limits.h	FLT_RADIX	constant integer macro
limits.h	FLT_ROUNDS	constant integer macro
math.h	HUGE_VAL	rvalue macro
limits.h	INT_MAX	constant integer macro
limits.h	INT_MIN	constant integer macro
limits.h	LDBL_DIG	constant integer macro
limits.h	LDBL_MAX_EXP	constant integer macro
limits.h	LDBL_MIN_EXP	constant integer macro
limits.h	LDBL_RADIX	constant integer macro
limits.h	LDBL_ROUNDS	constant integer macro
limits.h	LONG_MAX	constant integer macro
limits.h	LONG_MIN	constant integer macro
stdio.h	L_tmpnam	constant integer macro
assert.h	NDEBUG	macro reference
stdefs.h	NULL	constant pointer macro
limits.h	SCHAR_MAX	constant integer macro
limits.h	SCHAR_MIN	constant integer macro
stdio.h	SEEK_CUR	constant integer macro
stdio.h	SEEK_END	constant integer macro
stdio.h	SEEK_SET	constant integer macro
limits.h	SHRT_MAX	constant integer macro
limits.h	SHRT_MIN	constant integer macro
signal.h	SIGABRT	constant integer macro
signal.h	SIGFPE	constant integer macro
signal.h	SIGILL	constant integer macro
signal.h	SIGINT	constant integer macro
signal.h	SIGSEGV	constant integer macro
signal.h	SIGSTACK	added constant integer macro
signal.h	SIGTERM	constant integer macro
signal.h	SIGUSR1	constant integer macro
signal.h	SIGUSR2	constant integer macro
signal.h	SIG_DFL	constant pointer macro
signal.h	SIG_ERR	constant pointer macro
signal.h	SIG_IGN	constant pointer macro

Header File	*Identifier*	*Usage*
stdio.h	SYS_OPEN	constant integer macro
stdio.h	TMP_MAX	constant integer macro
limits.h	UCHAR_MAX	constant integer macro
limits.h	UINT_MAX	constant integer macro
limits.h	ULONG_MAX	constant integer macro
limits.h	USHRT_MAX	constant integer macro
stdio.h	_IOFBF	constant integer macro
stdio.h	_IOLBF	constant integer macro
stdio.h	_IONBF	constant integer macro
stdlib.h	abort	function or macro
math.h	abs	function or macro
math.h	acos	function or macro
time.h	asctime	function or macro
math.h	asin	function or macro
assert.h	assert	statement macro
ims.h	asmtdli	added function only
math.h	atan	function or macro
math.h	atan2	function or macro
stdlib.h	atof	old function or macro
stdlib.h	atoi	old function or macro
stdlib.h	atol	old function or macro
	auto	keyword
	break	keyword
stdlib.h	calloc	function or macro
	case	keyword
math.h	ceil	function or macro
	char	keyword
stdio.h	clearerr	function or macro
time.h	clock	function or macro
time.h	clock_t	type definition
	const	keyword
	continue	keyword
math.h	cos	function or macro
math.h	cosh	function or macro
ims.h	ctdli	added statement macro
time.h	ctime	function or macro
	default	keyword
	defined	#if or #elif operator
time.h	difftime	function or macro
ims.h	dlitc	added function reference
	do	keyword
	double	keyword
	else	keyword
	enum	keyword
math.h	erf	function or macro
math.h	erfc	function or macro
stdefs.h	errno	lvalue macro
stdlib.h	exit	function or macro
math.h	exp	function or macro
	extern	keyword
math.h	fabs	function or macro

Header File	*Identifier*	*Usage*
stdio.h	fclose	function or macro
stdio.h	feof	function or macro
stdio.h	ferror	function or macro
stdio.h	fflush	function or macro
stdio.h	fgetc	old function only
stdio.h	fgets	function or macro
	float	keyword
math.h	floor	function or macro
math.h	fmod	function or macro
stdio.h	fopen	function or macro
	for	keyword
stdio.h	fprintf	function or macro
stdio.h	fputc	old function only
stdio.h	fputs	function or macro
stdio.h	fread	function or macro
stdlib.h	free	function or macro
stdio.h	freopen	function or macro
math.h	frexp	function or macro
stdio.h	fscanf	function or macro
stdio.h	fseek	function or macro
stdio.h	ftell	function or macro
stdio.h	fwrite	function or macro
math.h	gamma	added function or macro
stdio.h	getc	function or unsafe macro
stdio.h	getchar	function or macro
stdlib.h	getenv	function or macro
stdio.h	gets	old function or macro
time.h	gmtime	function or macro
	goto	keyword
math.h	hypot	added function or macro
	if	keyword
	int	keyword
ctype.h	isalnum	function or macro
ctype.h	isalpha	function or macro
ctype.h	iscntrl	function or macro
ctype.h	isdigit	function or macro
ctype.h	isgraph	function or macro
ctype.h	islower	function or macro
ctype.h	isprint	function or macro
ctype.h	ispunct	function or macro
ctype.h	isspace	function or macro
ctype.h	isupper	function or macro
ctype.h	isxdigit	function or macro
math.h	j0	added function or macro
math.h	j1	added function or macro
setjmp.h	jmp_buf	type definition
math.h	jn	added function or macro
signal.h	kill	function or macro
math.h	ldexp	function or macro
time.h	localtime	function or macro
math.h	log	function or macro

Header File	*Identifier*	*Usage*
math.h	log10	function or macro
	long	keyword
setjmp.h	longjmp	function or macro
	main	function reference
stdlib.h	malloc	function or macro
string.h	memchr	function or macro
string.h	memcmp	function or macro
string.h	memcpy	function or macro
string.h	memset	function or macro
math.h	modf	function or macro
stdlib.h	onexit	function or macro
stdlib.h	onexit_t	type definition
stdio.h	perror	function or macro
math.h	pow	function or macro
stdio.h	printf	function or macro
stdefs.h	ptrdiff_t	type definition
stdio.h	putc	function or unsafe macro
stdio.h	putchar	function or macro
stdio.h	puts	function or macro
stdlib.h	rand	function or macro
stdlib.h	realloc	function or macro
	register	keyword
stdio.h	remove	function or macro
stdio.h	rename	function or macro
	return	keyword
stdio.h	rewind	function or macro
stdio.h	scanf	function or macro
stdio.h	setbuf	old function or macro
setjmp.h	setjmp	function or macro
stdio.h	setvbuf	function or macro
	short	keyword
signal.h	signal	function or macro
	signed	keyword
math.h	signgam	added rvalue macro
math.h	sin	function or macro
math.h	sinh	function or macro
stdefs.h	size_t	type definition
	sizeof	keyword
stdio.h	sprintf	function or macro
math.h	sqrt	function or macro
stdlib.h	srand	function or macro
stdio.h	sscanf	function or macro
	static	keyword
stdio.h	stderr	rvalue macro
stdio.h	stdin	rvalue macro
stdio.h	stdout	rvalue macro
string.h	strcat	function or macro
string.h	strchr	function or macro
string.h	strcmp	function or macro
string.h	strcpy	function or macro
string.h	strcspn	function or macro

Header File	*Identifier*	*Usage*
string.h	strlen	function or macro
string.h	strncat	function or macro
string.h	strncmp	function or macro
string.h	strncpy	function or macro
string.h	strpbrk	function or macro
string.h	strrchr	function or macro
string.h	strspn	function or macro
stdlib.h	strtod	function or macro
string.h	strtok	function or macro
stdlib.h	strtol	function or macro
	struct	keyword
	switch	keyword
stdlib.h	system	function or macro
math.h	tan	function or macro
math.h	tanh	function or macro
time.h	time	function or macro
time.h	time_t	type definition
time.h	tm	struct tag
stdio.h	tmpfile	function or macro
stdio.h	tmpnam	function or macro
ctype.h	tolower	function or macro
ctype.h	toupper	function or macro
	typedef	keyword
stdio.h	ungetc	function or macro
	union	keyword
	unsigned	keyword
stdarg.h	va_arg	rvalue macro
stdarg.h	va_end	statement macro
stdarg.h	va_list	type definition
stdarg.h	va_start	statement macro
stdio.h	vfprintf	function or macro
	void	keyword
	volatile	keyword
stdio.h	vprintf	function or macro
stdio.h	vsprintf	function or macro
	while	keyword
math.h	y0	added function or macro
math.h	y1	added function or macro
math.h	yn	added function or macro

Appendix D: Glossary

abbreviated declaration: A declaration you write with one or more components omitted, such as

```
static x; /* short for: static int x = 0; */

int a[2][2] = {0, 1, 2, 3};
          /* internal braces omitted */
```

access: To obtain or alter the value stored in a data object.

alphabetic character: An uppercase letter or a lowercase letter.

alphanumeric character: An uppercase letter, a lowercase letter, or a decimal digit.

application program: An executable program you write, as opposed to a "system program" that helps you develop and run your programs.

architecture: The set of machine instructions and data representations that are common to a family of computers, such as System/370.

argument: A data object allocated when you call a function. You provide the initial stored value for the data object with the "actual argument" expression you write in the function call. The function may alter the stored value of an argument. The argument data object is deallocated when the function returns.

argument level declaration: A declaration you write before the first left brace of a function body, to declare the arguments to that function. You may also write file level and block level declarations.

arithmetic types: A group of types which includes the integer and floating types.

array: An ordered sequence of data objects all of the same type.

array element: One member of an array. An array element is often designated by the address of the array combined with an element number, counting from zero. For example, **a[0]** is the first element of the array **a**.

assembler: A system utility that translates a text file, containing simple mnemonics describing machine instructions and data values, to an object code file. The compiler produces "assembly language" when it translates your source files.

You can also write assembly language source files directly.

assignment: Storing a new value in a data object. You can perform simple assignment, as in **x=y**, or use the current stored value to determine the new value, as in **x+=y**.

assignment compatible: Meeting the type requirements for the two operands of the assignment operator. An actual argument in a function call must also be assignment compatible with its corresponding function prototype argument. The expresssion in a return statement must be assignment compatible with the type returned by the function. Another constraint on types is that they be the "same type."

base type: That portion of the type you specify at the start of a declaration. Each declarator may provide additional type attributes along with its identifier, but the base type is shared among all declarators in a declaration.

binary stream: A stream you connect to a file for reading and writing arbitrary data. No conversion is performed between internal and external representations. You can also connect a "text stream" to a file.

bitfield types: A group of types that describe data objects consisting of contiguous groups of bits within an *int* component of a structure. You can specify a bitfield type only for a structure member. Bitfield types are integer types.

block level declaration: A declaration you write after the left brace of a *compound* statement, within a function body. You may also write file level and argument level declarations.

bounded string: A data object of type *array of char* whose defined content is specified by a number of elements *n*, or which ends with the first element containing a null character, whichever comes first. The C library also supports "buffers" and "strings."

buffer: A data object of type *array of char* whose defined content is specified by a number of elements *n*. The C library also supports "bounded strings" and "strings."

buffering: Reading and writing data in fixed size groups. Buffering can improve performance by increasing the number of bytes transferred on each call to the system services.

C compiler: A system utility that translates your C source files to assembly language, and then to object code files.

C library: A set of object code files that the standard linkage editor or loader can select from to obtain definitions of functions and data objects. You use the C library to perform input/output, storage allocation and deallocation, and a number of other useful services.

C runtime: The executable code that gets control when you invoke your C program. It initializes any data objects that must be defined at program startup, and opens the three standard

streams. It then calls the function **main** that you have defined. Upon program termination, the C runtime closes any open files and returns control to the operating system.

calling environment: The information needed to return control from a function call. You can save a calling environment by calling the function **setjmp**. A later call to the function **longjmp** designating the same calling environment causes execution to resume once again at the return from **setjmp**.

character constant: A token that specifies a value determined from the encoding of a character. Its type is *int*. You can write literal characters, such as **'a'**, or escape sequences, such as **'\n'**.

command line: The line of text you type to instruct the operating system to invoke an executable program. You can specify "command options" on the command line, which the C runtime converts to a sequence of strings. When it calls the function **main**, the C runtime provides arguments that designate these command option strings.

command option: One of a sequence of strings that the C runtime obtains from the command line used to invoke your program. One of the arguments to the function **main** gives the number of command options. Another argument designates the first string in an array of command option strings.

comment: A contiguous group of source characters between **/*** and ***/**. You use comments to separate tokens that the compiler might otherwise treat as a single token. You also use comments to make your source code more readable.

compilation: The source file you present to the compiler, plus any additional files that are included by **#include** preprocessor directives within the source text.

composite type: Any of the struct, union, or enum types. All composite types may have tags, and all must be defined by writing their "content" inside braces.

const type qualifier: The type qualifier **const**, that declares the base type or pointer type to be unmodifiable.

constant: A token that specifies the value of some data object type. You can determine both the value and the type by how you write a constant. The integer constant **123**, for instance, has the value 123 and the type *int*. There are integer constants, character constants, floating constants, and string constants.

constant expression: An expression that the compiler can evaluate where you write it in your source text. The compiler requires a "constant integer expression" for the size for the size of an array or bitfield, for instance. It requires a "constant expression" when you initialize a static data object of arithmetic type. It requires a "constant pointer expression" when you initialize a static data object of

pointer type.

content: The descriptive part of a struct, union, or enum declaration that you write inside braces.

conversion: Altering the representation of a value to that of another type.

data initializer: That portion of a declaration that specifies an initial value for a data object. A declarator followed by a data initializer constitutes a "defining instance" for its identifier. You can write constant expressions to specify the value that a data object with static lifetime assumes prior to program startup. You can write less restricted expressions to specify the value that a data object with dynamic lifetime assumes whenever control transfers to the start of the block in which it is declared.

data item: An expression, or list of constant expressions enclosed in braces, that you write to initialize a component of a data object.

data object: A contiguous sequence of bytes in memory used to hold values. You access a data object by writing an lvalue operand. The lvalue specifies a type, which determines what values are associated with the different bit patterns stored in the data object. Other languages combine the concepts of data object and lvalue into a single entity called a "variable."

data object types: A major group of types that describe data objects of known content. Each data object type occupies a known number of bits of storage, and can represent a known set of values. Other major groups of types include the function types and the incomplete types.

debugger: An optional facility you can specify when you compile a source file, that helps you locate errors in your program. The debugger lets you inspect and alter data objects while your program is executing.

declaration: A sequence of tokens you write to describe the functions, data objects, and type definitions for your program. A declaration consists of a storage class, a base type, and zero or more declarators. Each declarator provides an identifier and possibly additional type attributes. You may follow a declarator with a definition, which is a function body for a function or a data initializer for a data object. All of the tokens in your program belong either to preprocessor directives or declarations.

declarator: That portion of a declaration that provides additional type attributes and the identifier you wish to declare.

defining instance: That declaration which provides the definition for a function or data object. For a function, the declaration containing its function body is its defining instance. For a data object, the declaration containing its data initializer is its defining instance. If no data initializer is specified, then

at least one of the declarations, in one of the compilations, must be a "tentative definition" for that data object. In this case, the compiler provides a defining instance.

definition: That portion of a declaration that specifies a function body, for a function, or an initial value, for a data object. A declarator followed by a definition constitutes a "defining instance" of its identifier. Every function or data object must have a definition somewhere among the compilations you provide or within the C library. The compiler will provide a definition for a data object if you write at least one "tentative definition" for it.

designate: To provide the address of a function or data object. You designate a function to call it or to take its address. You designate a data object to access its stored value, alter its stored value, or take its address. A data object designator is an "lvalue."

domain error: A call to a C library function that specifies an input argument for which the function has no defined value. There are also "range errors."

dynamic lifetime: The period over which storage is allocated for a data object declared at argument level or block level, without the storage class **extern** or **static**. Your program allocates storage for a data object with dynamic lifetime when control transfers to the block in which you declared the data object. If you write a data initializer for the data object, the initial value is stored in the data object every time control transfers to the start of the block. Your program deallocates storage for the data object when control transfers out of the block. Data objects may also have "static lifetime."

enumeration constant: An identifier declared within the context of an enumeration type to have a constant value of the same type as the enumeration.

enumeration types: A group of types which includes all those declared with the **enum** keyword. An enumeration type is represented as an integer type. When you write an enumeration type, you specify one or more "enumeration constants," each of which has a constant value of the same type as the enumeration.

escape sequence: A contiguous group of characters within a character constant or a string constant that specifies a single character value. You can write a newline, for instance, as `'\n'`. You can also specify an octal value escape sequence, such as `'\17'`, or a hexadecimal value escape sequence, such as `'\xc5'`.

executable file: A file produced by the linkage editor or loader which you can run by typing a command line. Your program is an executable file.

expression: A sequence of tokens you write to call functions, compute values, and store values in data objects. Some expressions must be "constant expressions," since the compiler must be able to determine their values where you write them in your source text. The compiler translates other expressions to executable code so that your program carries out the action you specify when control transfers to the expression. The four classes of expressions are void expressions, lvalues, rvalues, and function designators.

external linkage: An attribute you specify for a function or data object to ensure that all compilations that declare the same identifier with external linkage refer to the same function or data object. You write a file level declaration with no storage class keyword, or a declaration at file or block level with the storage class **extern**, to specify external linkage. The external identifier derived from the identifier you write may have as few as seven significant characters, with no distinction between uppercase and lowercase letters. You can also specify "internal linkage" or "no linkage."

file level declaration: A declaration you write outside any other declaration. You may also write argument level and block level declarations.

file name: A string argument you specify to various C library functions to provide the external name of a file. The rules for writing file names vary among operating systems.

file pointer: An indicator associated with each stream connected to an open file that designates the next byte to read or write within the file. The file pointer is advanced on each read or write. You can also set the file pointer by calling the function **fseek**, and obtain its value by calling **ftell**.

floating constant: A token that specifies a value of some floating type. You can write fractional form, such as **12.4**, exponent form, such as **124e-1**, or both, such as **1.24e1**. You can also add a suffix to specify type, such as **12.4F**.

floating types: A group of types which can represent approximations to real numbers. Each type can represent a fixed number of bits of precision over a range of values that is large compared to the range of the integer types. Each type can represent negative numbers and the value zero as well.

format string: A string argument you specify to any of the "print" or "scan" functions in the C library to specify the number and nature of argument conversions to perform.

formatting: Converting between internal representation and printable text. You format input from a text stream by calling one of the "scan" functions in the C library. You format output to a text stream by calling one of the "print" functions in the C library.

function: A group of machine instructions that your program can execute. You transfer control to the start of the group by

calling the function. Control leaves the group when your program calls other functions or when it returns to the function that called it. You call a function by writing an expression. The function can return a value that becomes the value of the function call within the expression. All machine instructions the compiler generates from your source program are grouped into functions.

function designator: A class of expression that designates a function and has a type of the form *function returning T*. You use a function designator to call a function or to obtain its address. Other classes of expressions are lvalues, rvalues, and void expressions.

function prototype: A form of function declaration that specifies the number and types of all the arguments to a function call. You can also declare or define a function and specify no information about the number and types of its arguments.

function types: A major group of types that describe functions. Other major groups of types include the incomplete types and the data object types.

grouping: The act of determining what operands group with what operators in an expression. The compiler uses operator precedence to determine grouping if you do not fully parenthesize an expression.

header file: A file of source text provided with the C library that you can include in your compilations. Every C library function is declared in one of the header files, along with any needed type definitions and macro definitions.

identifier: A token that consists of an arbitrary sequence of uppercase letters, lowercase letters, underscore characters, and digits, beginning with anything but a digit. You use an identifier wherever you must name something in C. Characters after the first 31 are not necessarily significant when comparing two identifiers for equality. External identifiers may have as few as seven significant characters, with no distinction between uppercase and lowercase letters.

include file: A source file that you include as part of a compilation by writing the **`#include`** preprocessor directive as part of your source text. You can write your own include files, or include any of the "header files" provided with the C library.

incomplete types: A major group of types that describe data objects of unknown content, or the type *void*. You can use incomplete types wherever the compiler does not need to know the size of a data object, as when declaring a pointer to an incomplete type. You can declare an array of unknown content and specify its size by the data initializer you provide with the declaration. You can declare an argument that is an array of unknown content, since its type is changed to a pointer type. Other major groups of types include the function types and the data object types.

integer constant: A token that specifies a value of some integer type. You can write decimal integer constants, such as **123**, octal integer constants, such as **0400**, or hexadecimal integer constants, such as **0x2ce**. You can also add one or more suffixes to specify type, such as **127U**.

integer types: A group of types which can represent whole numbers. The "signed integer" types can represent a range of numbers from some negative value to some positive value. The "unsigned integer" types can represent a range of numbers from zero to some positive value.

internal linkage: An attribute you specify for a function or data object to ensure that all declarations that declare the same identifier with internal linkage, within one compilation, refer to the same function or data object. You write a file level declaration with the storage class **static** to specify internal linkage. You can also specify "external linkage" or "no linkage."

keyword: An identifier given special meaning by the compiler. You cannot create new keywords, and you cannot redeclare keywords.

label: An identifier you write in a *goto* statement to specify where control transfers within the function body. You write a "plain label" as a label identifier followed by a colon, before the statement that gets control. You can also write "*case* labels" and "*default* labels" within a *switch* statement.

linkage editor: A system utility that combines the object code files the compiler produces with those that the assembler and the C library produce to form an executable file.

lvalue: A class of expression that designates a data object and has a type. You use an lvalue to obtain the value stored in a data object, to obtain the address of the data object, or to store a value in the data object. Other classes of expressions are rvalues, function designators, and void expressions.

macro definition: A preprocessor directive of the form:

#define x *defn*

that defines the identifier **x** as a macro whose expansion is *defn*. You may also define macros that take arguments, whose expansions are substituted within the expansion of *defn*.

macro expansion: The replacement of a macro name, and any arguments, that you write in your source text with the expanded definition of the macro.

name space: A set of names, all of which must be distinct. All labels within a function body form a name space, for instance. You may use the same identifier as both a tag and a data object name, because these occupy separate name spaces.

narrowing conversion: A conversion from one arithmetic type to another that may result in loss of significant bits.

newline character: The character that causes the combined motion: carriage return, line feed. The compiler converts your source text to a character stream, recording the end of each text line with a newline character. The C library records the end of each line of a text file as a newline character, when you read a text stream. When you write a newline character to a text stream, the C library ends the current text record. You write the newline character as the escape sequence **\n** within a character constant or a string constant.

no linkage: An attribute you specify for a data object to ensure that only one declaration for that identifier refers to the data object. You write an argument or block level declaration with no storage class keyword, or with the storage class **auto** or **register**, to specify no linkage. You can also specify "external linkage" or "internal linkage."

null character: The character that terminates a string. The compiler appends a null character to every string constant. You write the null character as the escape sequence **\0** within a character constant or a string constant.

null pointer: A pointer whose value compares equal to the integer constant **0**. No function or data object in C has an address that compares equal to the null pointer.

object code file: A file produced by the assembler that captures the contribution to an executable file from a single C compilation or a single assembly language file. You use the linkage editor to combine object code files to produce an executable file.

operand: A constant, identifier, or subexpression that an operator operates upon within an expression. Examples of each are: the constant operand 3, the identifier operand **x**, and the subexpression operand ***(a+1)**.

operator: A token you write in an expression to specify what operation to perform upon one or more operands. You can negate a value, for instance, as in **-3**, call a function, as in **printf("hello\n")**, or store a value in a data object, as in **i=0**.

padding: Extra data bits added to a data object or the contents of a file that are not part of the value of any of the components you specify.

pointer types: A group of types which can represent the addresses of functions or data objects. There are pointers to functions, pointers to incomplete types, and pointers to data object types. You can use the value of a pointer type to designate a function or a data object, by writing an expression such as ***p** or **p[i]**.

precedence: The strength with which an operator groups operands relative to other operators. Higher precedence operators

group more strongly than lower precedence ones.

preprocessor: That portion of the compiler which carries out "preprocessor directives," signalled by lines beginning with the character #. You write preprocessor directives to include the contents of other files within a compilation, to conditionally skip over portions of your source text, and to define macros which are later expanded within your source text. The preprocessor rewrites your source text after it is grouped into tokens, before the tokens are grouped into declarations.

preprocessor directive: A source text line beginning with the character #. Each such line must match the pattern for one of the preprocessor directives defined by the preprocessor.

print function: Any of the C library functions that perform formatted output.

printable character: A character that has a visible graphic representation when written to a printer or a display device, or the space character.

program: An executable file that you specify by writing one or more source files, which are compiled to object code files and combined by the linkage editor.

program startup: The point in time after your program has been invoked, when control is first transferred to the function **main** that you provide.

program termination: The point in time after your program has transferred control to the C runtime for the last time.

punctuation character: A printable character other than an alphanumeric character or a space. C uses some of the punctuation characters to represent delimiters, separators, and operators.

range error: A call to a C library function that results in a computed value that cannot be represented in the required type. For floating types, you can get a range error either on underflow or on overflow. There are also "domain errors."

redeclaration: A declaration with external linkage or internal linkage that is in scope of a declaration for the same identifier that also has external linkage or internal linkage. A redeclaration describes the same function or data object as the earlier declaration.

representation: The set of bit patterns defined for a given data object type. Each valid bit pattern represents some value of that type. For a given type in C, all bit patterns occupy the same number of bits.

reentrant program A program that may be entered more than once, even if prior versions have not yet completed, as long as instructions and external data objects are not modified

during execution. Multiple copies of the text section of the same reentrant program may be shared by more than one user simultaneously.

reserved identifier: An identifier that has special meaning either to the compiler or to the C library. You may use a reserved identifier in a different name space from the one containing the special meaning.

rvalue: A class of expression that has a value and a type. It does not designate a data object. Other classes of expressions are lvalues, function designators, and void expressions.

same type: Having identical type, or type that is identical except for information unspecified in one of the two types. When you redeclare a function or data object, the redeclaration must have the same type as the earlier declaration. A data object pointer is assignment compatible with another data object pointer that points to data objects of the same type. Another constraint on types is that they be "assignment compatible."

scalar types: A group of types which includes the integer types, the floating types, and the pointer types. You can compare any of the scalar types against zero.

scan function: Any of the C library functions that perform formatted input.

scope: The region of source text over which a definition or declaration is associated with an identifier. A definition or declaration may be in scope but not "visible" because another definition or declaration masks it.

side effect: A change in the value stored in a data object, or a change in the state of a file, that occurs when your program evaluates an expression.

signal: A positive integer value you specify as an argument to the function **kill** to report an unusual event. You can call **kill** directly, or the C runtime environment can call it in response to an event it detects, such as an access to a storage address that is not part of your program. You call the function **signal** to alter the handling of signals.

signal handler: A function you write that gets control when **kill** reports a signal.

signed: Having negative, zero, and positive values.

standard error: A text stream that the C runtime connects to an open file before program startup. Your program can write error messages to this stream.

standard input: A text stream that the C runtime connects to an open file before program startup. Your program can read from this stream.

standard output: A text stream that the C runtime connects to an open file before program startup. Your program can write

to this stream.

statement: A sequence of tokens you write to perform actions and specify flow of control through your program. You write statements only inside a function body.

static lifetime: The period over which storage is allocated for a data object declared at file level, or with either the **extern** or **static** storage class. A data object with static lifetime has its initial value stored in it at program startup. Your program does not deallocate storage for it prior to program termination. Data objects may also have "dynamic lifetime."

storage alignment: A requirement imposed by the architecture on the values that the compiler may use for the addresses of data objects of a given type. A *short int*, for example, may be constrained to begin on an even byte boundary in memory.

storage allocation: The act of obtaining storage for a data object. The compiler allocates storage for data objects with static lifetime prior to program startup. Your program allocates storage for data objects with dynamic lifetime upon each entry to the block in which they are declared. You can also allocate storage at will by calling the library functions **calloc**, **malloc**, and **realloc**.

storage class: That portion of a declaration that provides linkage and lifetime for all declarators in the declaration. You write a storage class with one of the five keywords **extern**, **static**, **auto**, **register**, or **typedef**. The storage class **typedef** signals a type definition. You may apply **extern** and **static** to either functions or data objects. You may apply **auto** and **register** only to data objects.

The act of releasing storage for a data object. Your program will not deallocate storage for data objects with static lifetime until after program termination. It deallocates storage for data objects with dynamic lifetime upon each exit from the block in which they are declared. You can also deallocate storage you allocated by calling the library functions **calloc**, **malloc**, and **realloc**, by calling the library function **free**.

stream: A logical connection between a buffer and a file, controlled by a data object of type **FILE**. You connect a stream to a file by opening the file. You may then read or write the stream, which buffers input or output to the file. You break the connection by closing the file.

string: A data object of type *array of char* whose defined content ends with the first element containing a null character. The C library also supports "bounded strings" and "buffers."

string constant A token that specifies a value stored in memory as a data object of type *array of char*. The string constant **"ab\n"**, for example, has the four elements **'a'**, **'b'**, **'\n'**, and **'\0'**. The compiler always appends a null character to

a string constant.

struct types: A group of types which includes all those declared with the **struct** keyword. The content of a struct type consists of one or more structure members, each of which has a distinct name and may have a distinct data object type. The structure members specify component data objects that follow each other in memory.

structure member: An identifier declared within the context of a structure type to name a component of the structure type. All structure members must have data object types. You can specify a bitfield type only for a structure member.

structure types: A group of types which includes the struct types and the union types.

successful termination: A call to the function **exit** with an argument of zero.

tag: An identifier you declare as part of a struct, union, or enum type. You can declare a tag before you define the corresponding content. This is the only way to write structures that reference one another.

tentative definition: A declaration for a data object that ensures that the current compilation will provide a defining instance for the data object. If you write no data initializer for the data object, then the compiler will provide one with all zero values. You write a tentative definition at file level by omitting the storage class keyword from the declaration.

test expression: An rvalue of scalar type that is compared against zero. You write test expressions to control execution of statements, as in

```
if (test)
    statement
```

text line: A sequence of characters in a text stream represented internally with a newline character at the end of the line. Both the compiler and the C library treat a text stream as a sequence of text lines.

text stream: A stream you connect to a file for reading and writing printable text. Characters may be added, altered, or deleted to convert between the standard internal form for text and the representation used for a given file. You can also connect a "binary stream" to a file.

token: A contiguous group of characters in your source program that matches a predetermined pattern. Tokens form the elements of C declarations and preprocessor directives. The compiler groups all of the characters in your source program into tokens or into the whitespace and comments that separate tokens.

trigraph: A sequence of three characters you write in place of a C source character that your input device cannot generate. You can replace a left bracket, for instance, with the

trigraph **??(**.

type: The principal attribute of something you declare in C. The three major groups of types are function types, incomplete types, and data object types. Every constant you write also has some data object type.

type attribute: A sequence of tokens you add to a declarator to add type information. There are pointer type attributes, such as ***p**, array type attributes, such as **x[3]**, and function type attributes, such as **f(int)**.

type cast operator: An operator that converts a scalar rvalue to the type you specify in the operator. You write a type cast operator as a type name inside parentheses, such as **(double)x**.

type definition: A declaration with the storage class **typedef**, that defines an identifier to be a synonym for the type you specify.

type name: A declaration you write with no storage class, one declarator, no definition, and no identifier within the declarator. Examples are **double** or **int (*)()**. You write type names as part of type cast operators, for instance, or to declare arguments within a function prototype.

type qualifier: One of the keywords **const** or **volatile**. You can write a type qualifier as part of the base type in a declaration, or following the ***** pointer type attribute in a declarator.

union types: A group of types which includes all those declared with the **union** keyword. The content of a union type consists of one or more structure members, each of which has a distinct name and may have a distinct data object type. The structure members specify component data objects that overlap each other in memory.

unknown content: Describes an incomplete type that you can complete by providing additional information. You provide the size of an array to complete an "array of unknown content." You provide the structure member declarations to complete a "structure of unknown content." You cannot complete the type *void*.

unsigned: Having only zero and positive values.

unsuccessful termination: A call to the function **exit** with a nonzero argument.

value: The numeric meaning attached to a bit pattern by its associated data object type. The 32 bit pattern for an *int* with value 10, for instance, is a very small value as a *float* and an address in low memory as a *pointer to char*.

variable length argument list: An argument list on a function call that may differ in length from other argument lists in other calls on the same function. You declare a function that may accept variable length argument lists by writing a

prototype such as `int printf(char *fmt, ...)`. The notation `, ...` indicates that actual arguments from that point on behave as if there were no prototype in scope.

visibility: The region of source text over which a definition or declaration is in scope for an identifier, and the identifier is not masked by another definition or declaration.

void expression: A class of expression that has type *void*. It designates nothing and has no value. You evaluate a void expression for its side effects, such as calling a function or altering the value stored in a data object. Other classes of expressions are lvalues, rvalues, and function designators.

void type: The incomplete type *void*, which has no representation and which you cannot complete. You declare a function that returns no value as a *function returning void*. You declare a generic data object pointer as a *pointer to void*.

volatile type qualifier: The type qualifier **`volatile`**, that declares the base type or pointer type to be modifiable by agencies not obvious to the compiler.

whitespace: A contiguous group of spaces, horizontal tabs, vertical tabs, form feeds, carriage returns, and newlines. In your source code, you use whitespace to separate tokens that might otherwise be treated by the compiler as a single token. You also use whitespace to make your source code more readable. The C library enforces the same definition of whitespace in several functions.

widening conversion: A conversion from one arithmetic type to another that may result in an increase in the number of bits used to represent the value.

Index

preprocessor directive 7-11
#define preprocessor directive 3-1, 3-5, 3-7, 7-7, 8-2
#elif preprocessor directive 7-6
#else preprocessor directive 7-6
#endif preprocessor directive 7-4
#if constant 4-13
#if preprocessor directive 5-30, 7-5
#ifdef preprocessor directive 7-4
#ifndef preprocessor directive 7-5
#include file processing 7-2
#include files; search order 7-2
#include preprocessor directive 3-4, 7-2, 8-1, 10-6
#line preprocessor directive 7-11
#pragma preprocessor directive 1-4, 4-21, 5-19, 7-11
#undef preprocessor directive 3-7, 7-10, 8-2
31 bit addresses 4-18
31 bit addressing 1-2
<assert.h> header file 11-3
<ctype.h> header file 11-5
<ims.h> header file 11-19
<limits.h> header file 11-22
<math.h> header file 8-6, 11-25
<setjmp> header file 11-61
<signal.h> header file 11-64
<stdarg.h> header file 11-68
<stdefs.h header file 11-73
<stdefs.h> header file 8-6
<stdio.h> header file 8-4, 8-5, 9-1, 9-4, 9-7, 9-8, 11-74
<stdlib.h> header file 11-126
<string.h> header file 11-143
<time.h> header file 11-164
__FILE__ macro 7-9, 7-11
__LINE__ macro 7-9, 7-11
_pcblist data object 8-4
_pcblist macro 11-19
abbreviated declaration 3-8, 4-2, 4-24, D-1
abort function 8-7, 11-127
abs function 11-27
access D-1
acos function 11-28
add operator 5-22
additive operators 5-2, 5-22
address of operator 4-4, 5-21
addresses 2-8, 4-9, 5-13, 5-15
addressing operators 4-18, 5-2, 5-15
alert character 2-6, 9-3
allocation of registers 1-2
allocation of storage 4-4, 4-5
alphabetic character D-1
alphanumeric character D-1
ANSI standard 1-3, 1-4, 5-9, 7-9
ANSI; extensions 1-3, 1-4
application program D-1
architecture D-1
argument D-1
argument block 7-12
argument level declaration 3-3, 3-5, 3-6, 3-8, 4-2, 4-3, 4-16, 4-20, D-1
argument level redeclaration 4-6
argument list 4-20, 5-17
argument widening 5-18, 7-12, 11-69
arithmetic conversions 5-1, 5-5, 5-7, 5-27
arithmetic types 4-27, 5-14, D-1
array D-1
array element D-1
array type attribute 4-18, 4-19
array types 4-13, 4-19, 4-20, 4-23, 4-24, 4-26, 5-6, 5-14, 5-15
arrays; convert to pointers 3-5, 4-3
ASA control character 9-3
asctime function 11-165
asin function 11-29
asmtdli function 11-20
assembler D-1
assemblers 3-10
assembly language listings 1-2
assert function 11-4
assertion test 11-4
assignment D-1
assignment compatible 4-23, 5-12, 5-14, 5-17, 5-21, 5-27, 6-5, D-2
assignment conversions 5-1
assignment operators 5-2, 5-4, 5-27
asynchronous signals 4-17, 11-64
atan function 11-30
atan2 function 11-31
atof function 11-128
atoi function 11-129
atol function 11-130
auto data objects 11-62

auto keyword 4-2
backslash character 2-1, 2-2, 2-6, 2-7
backslash/newline sequence 2-2
base type 4-1, 4-7, D-2
binary input/output 9-1
binary stream D-2
binary streams 9-1, 9-3, 10-7, 11-85
bitfield types D-2
bitfields 4-10, 4-14, 4-15, 5-21
bitwise AND operator 5-25
bitwise exclusive OR operator 5-26
bitwise inclusive OR operator 5-26
bitwise NOT operator 5-19
bitwise shift operator 5-2, 5-23
block level declaration 3-3, 3-4, 3-5, 3-6, 3-7, 4-2, 4-4, 4-16, D-2
block level redeclaration 4-6
block scope 3-6, 4-7, 6-4
body of function 3-3
bounded string D-2
bounded strings 11-143
break statement 6-9, 6-10
broken vertical bar 2-10
buffer D-2
buffered input/output 9-1
buffering D-2
buffering strategy 9-5, 9-8, 9-9, 11-116
buffers 11-143
BUFSIZ macro 9-4, 11-74
byte order 10-7
C compiler D-2
C library 1-3, 1-5, 2-3, 3-10, 4-17, 4-22, 5-21, 5-23, 8-1, 8-5, 8-7, 9-1, 9-2, 9-5, 9-7, 9-8, 9-10, 11-1, D-2
C runtime 11-65, D-2
C runtime environment 3-10, 8-1, 11-19
C#INIT1 table 8-5
call C from other languages 7-12
call DL/I 11-20, 11-21
call other languages from C 7-12
calling environment 5-17, 6-5, 11-61, 11-62, D-3
calling sequence 7-12
calloc function 11-131
carriage return character 2-1, 2-6, 9-3, 11-14
case labels 6-2, 6-9
CC command 8-1
ceil function 11-32
char type 4-9
character constant D-3
character constants 2-4, 2-6
character set 2-1
character; alert 2-6
character; backslash 2-1, 2-2, 2-6, 2-7
character; carriage return 2-1, 2-6
character; escape 2-1, 2-6
character; form feed 2-1, 2-6
character; horizontal tab 2-1, 2-6
character; newline 2-1, 2-2, 2-6
character; null 2-7
character; question mark 2-6
character; space 2-1, 2-2
character; underscore 2-1, 2-3
character; unrecognized 2-10
character; vertical tab 2-6
character; virtual tab 2-1
character; whitespace 2-1, 2-2, 2-10
characters; printable 2-7
characters; punctuation 2-1
classes of expressions 5-5
classification of types 4-26
clearerr function 11-77
clock function 11-166
clock ticks 11-164, 11-166
closing files 8-5, 9-4, 9-6, 11-78
coding practice 2-5, 3-3, 4-2, 4-7, 4-8, 4-24, 5-3, 5-4, 5-10, 5-18, 8-6, 9-5, 9-6, 9-7, 9-8, 10-1
comma operator 5-2, 5-4, 5-15, 5-29
command line 8-3, 11-142, D-3
command option D-3
command options 8-3
comment D-3
comment delimiter 2-2, 2-7
comments 2-2, 2-7, 7-2, 7-9, 7-11, 10-3
comments; nest 2-2
commutative operators 5-3
compare identifiers 2-3, 3-10
comparing types 5-13
compilation D-3
compilations 1-6, 3-9, 4-22
compile time 4-1, 4-22, 5-1, 5-3,

5-29
compiler command option 7-2, 7-3
compiler error message 2-10, 3-8, 4-6, 4-16, 4-24, 5-29, 7-2, 7-12
compiler option 1-2, 1-3, 7-9
complex types 4-18, 4-25
composite type D-3
composite types 4-8, 4-9, 4-12
compound statement 6-1, 6-4, 6-9
concatenate string constants 2-7
conditional operator 5-2, 5-27
conditional preprocessor directive 7-1, 7-2, 7-4
conditional preprocessor directive group 7-2, 7-4
const keyword 4-8, 4-16
const pointers 4-19, 5-14
const type qualifier D-3
const types 4-16, 4-19, 4-21, 4-26, 5-6, 5-14, 10-5
constant 2-4, D-3
constant expression 2-6, 4-23, 4-24, 5-1, 5-3, 5-29, D-3
constant integer expression 2-5, 4-1, 4-13, 4-14, 4-19, 5-29, 6-9
constant pointer expression 5-30
constant suffixes; floating 2-6
constants; character 2-4, 2-6
constants; decimal 2-5
constants; decimal integer 2-4
constants; enumeration 4-1
constants; floating 2-4, 2-5
constants; hexadecimal integer 2-4
constants; integer 2-4
constants; negative 2-5, 2-6, 2-7
constants; string 1-3, 2-4, 2-7
constants; unrecognized 2-10
construct file names 7-2
construct string constants 2-7
construct tokens 7-10
content 4-12, 4-13, 4-14, 4-15, D-3
contexts for declaration 3-3, 4-7
continue statement 10-5
control character 11-8
conversion D-4
conversion specification 9-10, 11-88, 11-96
convert arrays to pointers 3-5, 4-3, 4-19, 4-21, 5-6, 5-15
convert data object pointers 5-11
convert function pointers 5-11
convert functions to pointers 3-5, 4-3, 4-21, 5-6
convert incomplete pointers 5-12
convert integer to pointer 5-10
convert lvalues to rvalues 5-6
convert pointers to integers 5-12
convert time 11-170
cos function 11-33
cosh function 11-34
creating files 9-5
ctdli function 1-3, 11-21
ctime function 11-167
data initializer 4-2, 4-4, 4-16, 4-17, 4-19, 4-22, 4-23, D-4
data item D-4
data items 4-23
data object D-4
data object definition 3-5, 4-22
data object designator 2-7, 5-6
data object pointer 4-18, 5-14, 5-15
data object types 4-7, 4-8, 4-19, 4-20, 4-24, 4-26, 5-14, D-4
data objects 1-5, 3-2, 3-5, 3-6, 3-7, 3-8, 4-1, 4-3, 4-6
date and time 11-164, 11-171
DDname 11-85
debugger D-4
debugging 1-2
decimal constants 2-5
decimal digits 2-1, 2-3, 11-9
decimal integer constants 2-4
declaration D-4
declaration contexts 3-3, 4-7
declaration; abbreviated 3-8, 4-2
declaration; argument level 3-3, 3-5, 3-6, 3-8, 4-2
declaration; block level 3-3, 3-4, 3-5, 3-6, 3-7, 4-2, 4-4
declaration; file level 3-3, 3-5, 3-7, 4-2, 4-5
declaration; implicit 3-7, 4-3
declaration; special tag 3-8
declarations 1-5, 3-1, 3-2, 4-1, 6-4, 6-9, 10-4
declarations; implementation of 3-1
declarations; linked 3-9
declarator D-4

declarators 4-1, 4-17, 4-22
default labels 6-2, 6-9, 10-5
defined operator 7-5
defining instance 4-22, D-4
definition D-5
definition; implicit 4-3
definition; tentative 4-3, 4-5, 4-6
definitions 1-5, 4-1, 4-2, 4-21, 4-22
definitions; type 3-7
delimit comments 2-2
delimiter; comment 2-7
delimiters 2-1, 2-9
designate D-5
difftime function 11-168
digits; decimal 2-1, 2-3
digits; hexadecimal 2-4
digits; octal 2-5, 2-7
directive; preprocessor 1-5
divide operator 5-22
DL/I 1-3
DL/I call; perform 11-21
DL/I; call 11-20
DL/I; modify argument with 11-20, 11-21
do/while statement 6-8, 6-10, 6-11, 10-5
domain error 8-6, 11-25, 11-29, 11-31, 11-42, 11-49, 11-50, 11-52, 11-55, 11-58, 11-59, 11-60, D-5
double quotation mark 2-6, 2-7, 7-2
double type 4-11
dynamic data initializers 4-22
dynamic lifetime 4-4, 4-5, 4-16, 4-22, 4-24, D-5
EBCDIC 2-1, 2-2, 2-6, 2-7, 2-10
editor; linkage 3-10
EDOM macro 8-6, 11-25
ellipsis 4-21
else/if chain 6-7, 10-4
empty lines in files 9-2
empty parentheses 5-17
empty text files 9-3
end of file 8-5, 9-4, 9-5, 9-9, 11-77, 11-79, 11-99
enforce grouping 5-3
enum keyword 4-12
enumeration constant D-5
enumeration constants 3-2, 3-5, 3-6, 3-7, 3-8, 4-1, 4-13, 5-30
enumeration types 3-2, 4-12, D-5
environment variables 11-134
environment; calling 11-62
environmental limits 11-23
EOF macro 8-5, 11-5, 11-74
equal to operator 5-25
equality operators 5-2, 5-24
ERANGE macro 8-6, 11-25
erf function 1-3, 11-35
erfc function 1-3, 11-36
errno macro 8-6, 11-25, 11-73, 11-106
error message; compiler 2-10, 3-8, 4-6
error messages 3-10
escape character 2-1, 2-6
escape sequence D-5
escape sequence; hexadecimal value 2-7
escape sequence; octal value 2-7
escape sequences 2-6, 2-7
executable file 1-5, 1-6, 8-1, 11-142, D-5
exit function 8-3, 8-7, 11-132
exp function 11-37
expand macros 3-1
expression D-5
expression statement 6-3
expression; constant 2-6
expression; constant integer 2-5, 4-1
expressions 5-1, 10-5
extensions to ANSI 1-3, 1-4
extern keyword 3-9, 4-2
external identifiers 3-10
external identifiers; length of 3-10
external linkage 3-10, 4-3, 4-5, 4-6, 4-22, 7-11, 10-6, D-6
fabs function 11-38
fclose function 9-6, 11-78
feof function 9-9, 11-79
ferror function 9-9, 11-80
fflush function 9-3, 9-8, 11-81
fgetc function 9-7, 11-82
fgets function 9-7, 11-83
file level declaration 3-3, 3-5, 3-7, 4-2, 4-5, 4-16, 4-23, D-6
file level redeclaration 4-6
file mode 11-84
file name D-6
file names 9-5, 11-84
file pointer 9-4, 9-8, 10-6, 11-99,

11-101, 11-113, D-6
file processing; #include 7-2
file record format 9-2
FILE type definition 8-4, 8-5, 9-1, 9-4, 11-74
file; executable 1-5, 1-6
file; object code 1-6
files; header 1-6, 3-5
files; source 1-5
files; VBS 11-100
fixed record file format 11-99
float type 4-11
floating constant D-6
floating constant suffixes 2-6
floating constants 2-4, 2-5
floating overflow 8-6, 11-25
floating types 4-8, 4-11, 4-23, 4-27, D-6
floating underflow 8-6, 11-25
floor function 11-39
flow of control 4-16, 5-17, 6-1, 6-3, 6-4, 6-9, 6-11
flushing buffers 9-6, 11-78, 11-81
fmod function 11-40
fopen function 7-3, 8-4, 9-5, 9-6, 11-84
for statement 6-8, 6-10, 6-11, 10-5
form feed character 2-1, 2-6, 9-3, 11-14
format source 2-2
format string 9-9, 11-88, 11-96, 11-107, 11-114, 11-118, 11-119, 11-123, 11-124, 11-125, D-6
formatted input 9-10, 11-96, 11-114
formatted input/output 9-1, 9-9
formatted output 9-11, 11-88, 11-107, 11-123, 11-124
formatting D-6
formatting; source 2-7
forward reference 3-8, 4-12
fprintf function 9-11, 11-88
fputc function 9-8, 11-92
fputs function 9-8, 11-93
fread function 9-7, 11-94
free function 11-133
freopen function 9-5, 11-95
frexp function 11-41
fscanf function 9-10, 11-96
fseek function 9-8, 11-99
ftell function 9-3, 9-8, 10-6, 11-101
function D-6
function body 3-3, 4-2, 4-21, 4-22
function calls 4-4, 4-16, 4-18, 4-21, 5-2, 5-4, 5-11, 5-12, 5-16, 6-1, 6-5, 7-12, 8-2
function definitions 3-5, 4-17, 4-20, 4-22, 4-25, 10-3
function designator 5-5, 5-16, 5-20, D-7
function prototype D-7
function prototypes 3-4, 4-20, 5-17, 10-2
function returning void types 4-9, 6-5
function to pointer conversion 3-5, 4-3
function type attribute 4-18, 4-20
function types 4-7, 4-20, 4-25, 4-26, D-7
functions 1-5, 3-2, 3-5, 3-6, 3-7, 3-8, 4-1, 4-5
functions; nesting 3-5
fwrite function 9-8, 11-102
gamma function 1-3, 11-42
getc function 9-7, 11-103
getchar function 9-7, 11-104
getenv function 11-134
gets AND operator 5-28
gets divided operator 5-28
gets exclusive OR operator 5-28
gets function 9-7, 11-105
gets inclusive OR operator 5-28
gets left shifted operator 5-28
gets multiplied operator 5-27
gets operator 5-27
gets remainder operator 5-28
gets right shifted operator 5-28
gets subtracted operator 5-28
gmtime function 11-169
goto statement 3-2, 3-7, 3-8, 6-2, 6-4, 6-11, 10-5
graphic character 11-10
greater than operator 5-24
greater than or equal to operator 5-24
Greenwich Mean Time 11-164, 11-169
grouping 5-24, D-7
grouping of operators 5-1
grouping tokens 2-10
grouping; enforce 5-3
header file D-7
header files 1-6, 3-5, 7-10, 8-1, 10-2, 10-6, 11-1
hexadecimal digits 2-4, 11-16

hexadecimal integer constant 2-4
hexadecimal value escape sequences 2-7
horizontal tab character 2-1, 2-6, 9-3, 11-14
HUGE_VAL macro 8-6, 11-25
hypot function 1-3, 11-43
identifier D-7
identifiers 1-5, 2-1, 2-3, 3-1, 4-1
identifiers; compare 2-3, 3-10
identifiers; external 3-10
identifiers; length of 2-3
identifiers; reserved 2-3
if statement 6-5
if/else statement 6-6
implement declarations 3-1
implicit declaration 3-7, 4-3, 4-20, 5-16, 5-18
implicit definition 4-3
IMS 1-2, 1-3
IMS environment 8-4
IMS; _pcblist macro 11-19
IMS; call DL/I 11-20, 11-21
IMS; invoke program from 8-4
IMS; link for 11-19
IMS; signal handling 8-4
include file D-7
include file processing 7-2
include files; search order 7-2
incomplete pointer 5-14
incomplete text line 7-2, 10-6
incomplete types 4-7, 4-9, 4-12, 4-19, 4-20, 4-22, 4-23, 4-24, 4-26, D-7
indirect on operator 4-18, 5-20
inherited linkage 4-5, 4-6
initializer; data 4-2, 4-4, 4-16, 4-17, 4-19, 4-22, 4-23, D-4
int type 4-10
integer constant 2-4, D-7
integer constant suffixes 2-4
integer constant; hexadecimal 2-4
integer constant; octal 2-5
integer constants 2-4, 4-13
integer expression 4-13
integer overflow 11-141
integer types 4-8, 4-9, 4-23, 4-27, D-8
integer; two's complement 2-5
integer; unsigned long 2-5
interactive terminals 8-4, 9-3, 9-8
internal linkage 3-9, 4-3, 4-6, 4-22, 10-6, D-8
invoke program from IMS 8-4
invoking programs 1-6, 8-1, 9-5, 11-132, 11-142
isalnum function 11-6
isalpha function 11-7
iscntrl function 11-8
isdigit function 11-9
isgraph function 11-10
islower function 11-11
isprint function 11-12
ispunct function 11-13
isspace function 11-14
isupper function 11-15
isxdigit function 11-16
j0 function 1-3, 11-44
j1 function 1-3, 11-45
JCL 1-2
jn function 1-3, 11-46
keyword D-8
keywords 2-4, 3-1, 3-5, 3-6, 3-7, 3-9, 10-4
kill function 11-64
label D-8
labels 3-2, 3-5, 3-6, 3-7, 3-8, 6-2, 6-11, 10-5
ldexp function 11-47
leading underscore character 2-3
left shift operator 5-23
length of byte 11-23
length of bytes 4-9, 9-1
length of external identifiers 3-10, 10-6
length of identifiers 2-3
length of macro expansion 7-8
length of records 9-2
length of source line 2-2
length of strings 11-156
length of text line 2-2
less than operator 5-24
less than or equal to operator 5-24
letters; lowercase 2-1, 2-3, 3-10
letters; uppercase 2-1, 2-3, 3-10
lexical order 11-143, 11-146, 11-152, 11-158
lines; physical text 2-2
lines; source 2-2
link for IMS 11-19
linkage 3-1, 3-9, 4-1
linkage editor 1-6, 3-10, 8-1, D-8
linkage editor error messages

3-10
linkage; external 3-10, 4-3, 4-5, 4-6
linkage; inherited 4-5, 4-6
linkage; internal 3-9, 4-3, 4-6
linked declarations 3-9
listings; assembly language 1-2
loader 1-6, 3-10, 8-1
local time 11-164, 11-167, 11-170
localtime function 11-170
log function 11-49
log10 function 11-50
logarithm; compute 11-49
logical and conditional operators 5-26
logical AND operator 5-26
logical NOT operator 5-19
logical operators 5-2, 5-4
logical OR operator 5-26
long double type 4-12
long type 4-11
longjmp function 11-62
lowercase letters 2-1, 2-3, 3-10, 11-6, 11-7, 11-11, 11-17, 11-18
lvalue 5-5, 5-6, 5-15, 5-16, 5-20, 5-21, D-8
macro arguments in string constants 1-3, 7-9
macro definition D-8
macro definition; remove 3-6
macro expansion 3-1, 7-1, 7-2, 7-4, 7-7, D-8
macros 1-5, 3-1, 3-4, 3-6, 3-7, 7-7, 8-2, 9-7, 9-8, 10-2
macros with arguments 7-7
main function 3-10, 8-3, 8-7, 11-136
maintenance 5-18, 10-1
malloc function 11-135
masking names 3-7, 3-9, 4-7, 11-1, 11-76
member names 7-3
members of a structure 3-2, 3-5, 3-6, 3-7, 3-9, 4-1
memchr function 11-145
memcmp function 11-146
memcpy function 11-148
memset function 11-149
minus operator 5-20
modf function 11-51
modifiable lvalue 5-6, 5-19, 5-27, 11-73
modify argument with DL/I 11-20, 11-21
modulus arithmetic 5-10
multiple dimension arrays 5-15
multiple file programs 10-6
multiplicative operators 5-2, 5-22
multiply operator 5-22
mutually referring structures 3-8, 4-14
MVS 1-2, 7-3, 8-4, 9-2, 9-3, 11-84, 11-101, 11-166
MVS/XA 1-2, 7-3, 9-2, 9-4, 11-84, 11-101, 11-166
name space D-8
name spaces 3-1, 3-6
names of preprocessor directives 3-1
names; mask 3-7, 3-9, 4-7
names; scope of 3-1, 3-6
narrowing conversion D-8
narrowing conversions 5-9
natural log 11-49
negative constants 2-5, 2-6, 2-7
nesting comments 2-2
nesting data initializers 4-24
nesting function calls 5-17, 11-61
nesting functions 3-5
nesting if statements 6-6
nesting include files 7-2
nesting switch statements 6-9
newline character 2-1, 2-2, 2-6, 9-1, 9-2, 11-14, D-9
no linkage 3-10, 4-4, 4-5, 4-6, 4-22, D-9
not equal to operator 5-25
null character 2-7, 4-24, 11-144, D-9
null characters 9-2, 9-3
NULL macro 11-73
null pointer 5-10, 8-3, D-9
null statement 6-4
null string constant 2-7
object code file 1-6, D-9
object code files 8-1
objects; data 3-2
octal digits 2-5, 2-7
octal integer constant 2-5
octal value escape sequence 2-7
offset 11-101
offsets 4-13, 4-15, 5-16, 9-9, 11-99
onexit function 11-136
opening files 9-4, 11-84
operand D-9

operator D-9
operator grouping 4-18
operator; address of 4-4
operator; sizeof 3-4
operators 2-1, 2-9, 5-1
operators; type cast 3-4
order of evaluation 5-4, 5-26, 5-29
OS calling sequence 1-4, 4-21, 5-19, 7-11
OS linkage 7-12
overflow 11-27
overlapping storage 4-15, 4-16, 11-148, 11-154, 11-159
overloaded keywords 4-7
packing bitfields 4-14
padding 4-9, 4-13, 4-14, 4-15, 9-2, 9-3, 11-86, 11-99, D-9
partitioned dataset members 7-3
PCB 11-19
perror function 11-106
physical text lines 2-2
plain label 6-2
plus operator 5-3, 5-20
point at member operator 5-2, 5-16
pointer arithmetic 5-12, 5-22, 5-23
pointer conversions 5-1, 5-10, 5-21
pointer to incomplete type 4-19
pointer to void type 4-9, 4-18, 5-12, 5-14, 5-27, 11-143
pointer to void types 4-26
pointer type attribute 4-18
pointer types 4-18, 4-23, 4-26, D-9
pointer; convert function to 4-3
pointers; convert arrays to 3-5, 4-3
pointers; convert functions to 3-5
portability 2-3, 2-7, 2-8, 4-16, 4-21, 5-10, 5-12, 5-14, 5-18, 7-9, 8-6, 10-6
positioning streams 9-4, 9-8, 11-99, 11-101
postdecrement operator 5-4, 5-20
postincrement operator 5-4, 5-20
pow function 11-52
pragmas 7-11
precedence 5-2, D-9
predecrement operator 5-4, 5-19
predefined macros 7-9
preincrement operator 5-4, 5-19
preprocessor 1-5, 3-7, 7-1, D-9
preprocessor directive 1-5, 3-1, 7-1, D-10
preprocessor directive names 3-1
preprocessor directive; #pragma 1-4
print function D-10
printable character 2-7, 9-1, 11-8, 11-10, 11-12, D-10
printf function 4-21, 5-5, 9-11, 11-107
process #include files 7-2
program D-10
program communication block 11-19
program listings 1-2
program startup 4-4, 4-16, 4-22, 8-3, 8-5, 11-67, D-10
program structure 1-5
program termination 4-4, 4-16, 8-3, 8-7, 9-6, 11-64, 11-127, 11-132, 11-136, D-10
programs; invoking 1-6
ptrdiff_t type definition 5-23, 10-7, 11-73
punctuation character D-10
punctuation characters 2-1, 11-13
putc function 9-8, 11-108
putchar function 9-8, 11-109
puts function 9-8, 11-110
question mark character 2-1, 2-6
rand function 11-137
random numbers 11-137, 11-139
range error 8-6, 11-25, 11-34, 11-37, 11-42, 11-43, 11-44, 11-45, 11-48, 11-49, 11-50, 11-52, 11-54, 11-56, 11-58, 11-59, 11-60, 11-140, 11-141, D-10
read/write error 8-6, 9-4, 9-5, 9-9, 11-77, 11-80
readability 2-7, 4-7, 4-8, 4-24, 4-25, 5-3, 10-1
reading complex attributes 4-18
reading streams 9-4, 9-7
reading type names 4-26
realloc function 11-138
record format 11-86
record length 11-86
recursion 5-17
redeclaration D-10
redeclaration; block level 4-6
redeclaration; file level 4-6
redeclarations 3-9, 4-5, 4-7,

4-19, 4-20, 4-21
redefine macros 3-7, 7-7
redirect standard input 8-4
redirect standard output 8-4
redundant parentheses 4-18, 4-25, 5-3, 5-5
reentrant code 1-2
reentrant program 2-8, 8-5, D-10
register allocation 1-2
register data objects 4-4, 4-5, 5-21, 11-62
register keyword 4-2, 4-21
regrouping 5-1, 5-3, 5-20, 5-22, 5-25
regrouping; defeat 5-3
relational operators 5-2, 5-24, 10-5
remainder operator 5-22
remove function 11-111
remove macro definitions 3-6, 7-10, 8-2
removing files 11-111
rename function 11-112
renaming files 11-112
representation 4-8, 4-9, 4-12, 5-11, 5-12, D-10
reserved identifier D-11
reserved identifiers 2-3, 4-7, 7-1, C-1
return statement 5-17, 6-1, 6-5, 10-5
rewind function 9-8, 11-113
rewrite variable length record format files 9-2
right shift operator 5-23
RLINK program 8-5
rules; unsignedness preserving 1-3
rules; value preserving 1-3
runtime environment 3-10, 8-1, 11-19
runtime error message 8-7, 11-106
rvalue 5-5, 5-6, 5-15, 5-16, 5-17, 5-19, 5-20, 5-22, 5-23, 5-24, 5-25, 5-26, 5-27, 5-29, 6-1, D-11
same type D-11
same types 3-10, 4-8, 4-21, 5-13, 5-14
scalar types 4-23, 4-26, D-11
scan function D-11
scanf function 9-10, 11-114
scope D-11
scope of names 3-1, 3-6, 4-20, 4-21, 7-7
scope; block 4-7
search modifier 7-2
search order for #include files 7-2
select member operator 5-2, 5-16
separate tokens 2-2, 2-9, 2-10
separators 2-1, 2-9, 11-163
sequences; escape 2-6
setbuf function 9-9, 11-115
setjmp function 11-63
setvbuf function 9-9, 11-117
short type 4-10
side effect D-11
side effects 5-1, 5-4, 5-19, 5-26, 5-27, 6-1, 8-2, 9-7, 9-8, 11-76
SIGABRT signal 8-7, 11-66, 11-127
SIGFPE signal 11-66
SIGILL signal 11-66
SIGINT signal 11-66
signal 11-127, D-11
signal function 1-3, 8-4, 8-7, 11-67
signal handler 4-16, 4-17, 11-64, D-11
signal handling under IMS 8-4
signals 4-16, 11-64
signed D-11
signed bitfield types 4-14
signed char type 4-10
signed int type 4-10, 4-14
signed integers 4-9, 5-10
SIGSEGV signal 11-66
SIGTERM signal 11-66
SIGUSR1 signal 1-3, 11-66
SIGUSR2 signal 1-3, 11-66
sin function 11-53
single quotation mark 2-6
sinh function 11-54
size of data objects 4-9, 4-12, 4-14, 4-19, 4-20, 4-24, 5-11, 5-12, 5-16, 5-21, 10-7
size_t type definition 5-21, 10-7, 11-73
sizeof operator 3-4, 5-21
solid vertical bar 2-10
source files 1-5, 7-1, 10-1
source formatting 2-2, 2-7, 10-1
source lines 2-2
space character 2-1, 2-2, 9-2, 11-14
special tag declaration 3-8, 4-12
SPIE system service 11-65

sprintf function 9-11, 11-118
sqrt function 11-55
srand function 11-139
sscanf function 9-10, 11-119
standard error D-11
standard error stream 8-4, 9-5, 11-4
standard input D-11
standard input stream 8-4, 9-5
standard output D-11
standard output stream 8-4, 9-5
startup of program 4-4
statement D-11
statement; goto 3-2, 3-7, 3-8
statements 6-1
static data initializers 4-22, 5-30
static keyword 4-2
static lifetime 4-4, 4-5, 4-16, 4-22, 4-24, 5-30, 10-2, D-12
STAX system service 11-65
stderr macro 8-4, 9-5, 11-75
stdin macro 8-4, 9-5, 11-75
stdout macro 8-4, 9-5, 11-75
storage alignment 4-9, 4-13, 4-15, 5-11, D-12
storage allocation 4-4, 4-5, 4-8, 4-9, 4-22, 5-17, 6-4, 6-9, 9-9, 11-78, 11-115, 11-117, 11-131, 11-133, 11-135, 11-138, D-12
storage class 4-1, 4-2, 4-8, 4-23, D-12
storage overlap 5-13, 11-148, 11-154, 11-159
storing in string constants 1-4, 2-8
strcat function 11-150
strchr function 11-151
strcmp function 11-152
strcpy function 11-154
strcspn function 11-155
stream D-12
string 9-10, 9-11, D-12
string constant D-12
string constant concatenation 2-7
string constant; null 2-7
string constants 1-2, 1-3, 2-4, 2-7, 4-24, 7-8
string constants; construct 2-7
string constants; macro arguments 1-3
string constants; storing in 1-4, 2-8
strings 8-3, 9-5, 11-143
strlen function 11-156
strncat function 11-157
strncmp function 11-158
strncpy function 11-159
strpbrk function 11-160
strrchr function 11-161
strspn function 11-162
strtod function 11-140
strtok function 11-163
strtol function 11-141
struct keyword 4-13
struct tags 4-24
struct types 4-12, 4-13, D-12
structure expressions 5-17, 5-27, 5-29
structure member D-13
structure members 3-2, 3-5, 3-6, 3-7, 3-9, 4-1, 4-13, 4-14, 4-15, 4-16, 4-24, 5-16
structure of programs 1-5
structure rvalues 5-16
structure types 3-2, 4-12, 4-23, 4-26, 5-13, 5-16, D-13
structures; referring mutually 3-8
subexpressions 5-1, 5-5
subscript 4-18, 5-2, 5-15
subtract operator 5-23
successful termination 8-7, 11-132, D-13
suffixes; floating constant 2-6
suffixes; integer constant 2-4
support for debugging 1-2
switch statement 6-2, 6-4, 6-9, 6-10, 10-5
system function 8-4, 11-142
System V; UNIX 1-3
System/370 2-5, 2-6, 2-7, 4-4, 4-9, 4-10, 4-11, 4-12, 4-14, 4-15, 4-18, 5-7, 5-9, 5-10, 5-11, 5-12, 5-19, 5-21, 5-23, 6-4, 10-2, 10-3, 11-27, 11-134
tag D-13
tags 3-2, 3-5, 3-6, 3-7, 3-8, 4-1, 4-12, 4-13, 4-15
tan function 11-56, 11-57
tanh function 11-57
temporary files 11-120, 11-121
tentative definition 4-3, 4-5, 4-6, 4-22, D-13
termination of program 4-4
test expression 6-1, 6-5, 6-7,

6-8, 11-4, D-13
text line D-13
text line; length of 2-2
text lines 7-1, 9-1
text stream D-13
text streams 8-4, 9-1, 10-6, 11-85
time conversion 11-170
time function 11-171
tmpfile function 11-120
tmpnam function 11-121
token D-13
tokens 2-1, 2-4, 7-1, 7-8, 7-9, 11-163
tokens; group 2-10
tokens; separate 2-2, 2-9
tolower function 11-17
toupper function 11-18
trigraph D-13
trigraphs 2-1
truncation 5-9, 5-10
TSO 1-2, 8-3, 11-84
two's complement integer 2-5, 4-14
type 2-4, 4-7, 5-5, D-13
type attribute D-14
type cast operator D-14
type cast operators 3-4, 5-10, 5-12, 5-14, 5-21, 5-30
type definition D-14
type definitions 3-2, 3-5, 3-7, 3-8, 4-1, 4-2, 4-25, 10-2, 11-69
type name D-14
type names 3-4, 4-20, 4-25, 5-21, 11-69
type qualifier D-14
type qualifiers 4-8, 4-16, 4-19, 4-26
type specifiers 4-2, 4-8, 4-25
typedef keyword 4-2, 4-25
types; enumeration 3-2
types; structure 3-2
unary operators 5-2, 5-19
underscore character 2-1, 2-3
unexpected errors 5-9
ungetc function 9-7, 11-122
union keyword 4-15
union types 4-12, 4-15, 4-24, D-14
UNIX System V 1-3
unknown content 4-7, 4-8, 4-12, 4-14, 4-19, 4-22, 4-23, 4-26, 5-13, 5-15, D-14
unnamed bitfields 4-14
unrecognized character 2-10
unrecognized constant 2-10
unsafe macro 9-7, 9-8, 11-76
unsigned D-14
unsigned bitfield types 4-14
unsigned char type 4-10
unsigned conversions 5-10
unsigned int type 4-10
unsigned integers 4-9, 4-14, 5-10
unsigned long integer 2-5
unsigned long type 4-11
unsigned short type 4-10
unsignedness preserving rules 1-3
unsuccessful termination 8-7, 11-127, 11-132, D-14
update streams 11-85
uppercase letter 11-6, 11-7, 11-15, 11-17, 11-18
uppercase letters 2-1, 2-3, 3-10, 11-84
va_arg function 11-69
va_end function 11-71
va_start function 11-72
valid pointers 5-12, 5-20
value D-14
value preserving rules 1-3
values 4-8, 4-9, 5-1
Variable Block Span files 11-100
variable length argument list 4-21, 9-9, 11-68, 11-88, 11-107, 11-114, 11-118, 11-119, 11-123, 11-124, 11-125, D-14
variable length record file format 11-99
variable length record format files 9-2
VBS files; restrictions on seeking 11-100
vertical tab character 2-6, 11-14
vfprintf function 9-11, 11-123
virtual tab character 2-1
visibility 3-1, 3-7, 3-9, 4-5, 4-7, D-15
VM/CMS 1-2, 7-2, 9-2, 9-3, 11-84, 11-101, 11-121, 11-166
void expression D-15
void expressions 5-4, 5-5, 5-6, 5-17, 5-27, 5-29, 6-1, 6-3
void type 4-7, 4-8, 4-9, 4-20, 4-26, 5-4, 5-5, D-15
volatile keyword 4-8, 4-16
volatile pointers 4-19, 5-14
volatile type qualifier D-15

volatile types 4-17, 4-21, 4-26, 5-14
vprintf function 9-11, 11-124
vsprintf function 9-11, 11-125
while statement 6-7, 6-10, 6-11
whitespace D-15
whitespace character 2-1, 2-2, 2-10, 7-7, 11-14
widening arguments 5-18
widening conversion D-15
widening conversions 5-7
widening order 5-7
widening to int 5-7, 5-18, 7-12, 11-69
wraparound arithmetic 5-10
writing streams 9-4
writing type names 4-25
y0 function 1-3, 11-58
y1 function 1-3, 11-59
yn function 1-3, 11-60

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter **with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.**

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s): Comment(s):

Please contact your nearest IBM branch office to request additional publications.

Name _____

Company or Organization _____

Address _____

_____

SC09-1129-01

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 6R1T
180 Kost Road
Mechanicsburg, Pennsylvania 17055

Fold and tape | Please Do Not Staple | Fold and tape

IBM®