
INTRODUCTION TO THE C PROGRAMMING LANGUAGE

Mark C. Allman

Revised August, 1994

Copyright ©1994 by Mark C. Allman

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the author.

CONTENTS

Preface	10
1 PROGRAM & LANGUAGE OVERVIEW	11
1 Introduction	11
2 Capability vs. Syntax	11
3 C Language Capabilities	12
3.1 C Compiler Pre-Processor	12
3.2 Derived Data Types Built From Built-In Data Types	12
3.3 Assignment Statements	13
3.4 Conditional Execution	14
3.5 Repetitive Execution	14
3.6 Recursive Execution	14
3.7 External Definition of Subroutines	15
3.8 Address and Bit Operations	15
3.9 Formatted and Unformatted Input/Output	15
4 Basic Structure	15
4.1 Program Components	16
4.2 Structuring Programs using Components	18
5 Summary	20
2 DATA TYPES	23
6 Introduction	23
7 Bits and Bytes	23
8 Built-In Types	25

8.1	Character Data	25
8.2	Integer Data	25
8.3	Floating Point Data	25
8.4	Void Data Type and its Uses	26
9	Scope of a Variable	26
10	Data Type Qualifiers	27
11	Derived Types	29
11.1	Structures	29
11.2	Defining New Type Names (typedef)	31
11.3	Unions	32
11.4	Enumeration	33
11.5	Arrays	34
12	Pointers	35
12.1	Call by Value and Call by Reference	36
12.2	Array Names as Pointers	37
12.3	Arrays of Pointers	38
12.4	Void Pointer Type	39
12.5	Pointers to Functions	40
13	Rules for Defining Variables	42
14	Constants	43
14.1	Character Constants	43
14.2	String Constants	44
14.3	Integer Constants	44
14.4	Floating-Point Constants	45
15	Logical Values and Boolean Values	45
3	C OPERATORS	47
16	Introduction	47

17	Arithmetic Operators	48
18	Logical Operators	49
19	Boolean Operators	50
20	Bitwise Operators	51
21	Address Operators	53
22	Special Operators	54
23	Operator Precedence	55
4	C LANGUAGE STATEMENTS	57
24	Introduction	57
25	Statement Format	57
25.1	Simple Statements	57
25.2	Compound Statements	58
25.3	Null Statement	58
26	Assignment Statement	59
27	Statement Format Style	59
28	Calling Subroutines	59
29	Control Flow Statements	60
29.1	If	60
29.2	Switch	61
29.3	While	63
29.4	Do..While	64
29.5	For	65
30	Control Flow Modifiers	66
30.1	Break	66
30.2	Continue	67
30.3	Return	67
5	C INPUT/OUTPUT LIBRARY	69

31	Introduction	69
32	Files and Devices	69
33	Character I/O	69
34	Formatted (Text) I/O	69
35	Unformatted (Binary) I/O	69
6	C STANDARD LIBRARIES	71
36	Introduction	72
37	String Functions	72
38	Math Functions	72
39	Memory Management Functions	72
40	Data Conversion Functions	72
41	Date/Time Functions	72
42	System and Program Exit Functions	72
43	Character Conversion and Testing Functions	72
44	Sorting and Searching Functions	72
45	Miscellaneous Functions	72
A	C PRE-PROCESSOR	73
A.1	Macro Definition	73
A.2	Defining Constants	75
A.3	Removing Definitions	75
A.4	Including Files	76
A.5	Conditional Compilation Directives	76
A.6	Additional Directives	77
6.1	Line	78
6.2	Error	78
6.3	Pragma	78

B ASCII CHARACTER SET 79

C C OPERATOR TABLE 81

Bibliography 82

LIST OF FIGURES

1.1	Language Capability Comparison	12
1.2	Sample C Input/Output Routines	16
1.3	Program Built from Components	21
2.1	Memory Terms and Sizes	24
2.2	C Built-In Data Types	25
2.3	Signed/Unsigned Value Ranges	28
2.4	Special Character Sequences	44
3.1	Arithmetic Operators	48
3.2	Logical Operators	50
3.3	Boolean Operators	50
3.4	Bitwise Operators	51
3.5	Sequence of Lights Example	53
3.6	Address Operators	53
3.7	Special Operators	56
A.1	Pre-Processor Directives	73

Preface

The purpose of this set of notes is to introduce the C programming language and familiarize users (and perhaps developers) with the language style employed in the STAMPS (Spacecraft Trajectory Analysis and Mission Planning Simulation) program.

M. A., September, 1993

Chapter 1

PROGRAM & LANGUAGE OVERVIEW

1 Introduction

We discuss in this chapter the principal components and capabilities of the C language. These components can be thought of as “building blocks” with which one constructs the tool (program) for the task at hand. The analogy of programs to tools in a manufacturing or automobile shop helps to put programs in perspective. We have custom tasks that we need to build custom tools to perform.

2 Capability vs. Syntax

We have postponed the discussion of the syntax of C programming constructs and instead in this chapter outline some typical C program components. *Syntax* is defined by Webster’s Dictionary as “...the way in which words are put together to form phrases, clauses, or sentences.” [7] As an example, the syntax of a statement setting a equal to $b + c$ in Pascal and C looks like

$$\begin{array}{ll} \text{C} & \rightarrow a = b + c; \\ \text{Pascal} & \rightarrow a := b + c; \end{array}$$

It is important to separate the syntax and purpose of program statements. Every programming language provides a mechanism to define variables and perform computations. How one sets up the statement is a question of syntax. That languages provide these mechanisms is a question of capability. Programming languages of the same generation (C, Pascal, Fortran, Basic, PL-1, RPG II and Cobol are 3rd generation languages) typically have a common set of capabilities. A general rule-of-thumb is that the first programming language learned is usually the most difficult. The reason is that after learning the first language one has an idea of what capabilities exist. Learning new languages then reduces to learning syntax¹.

¹Usually each language has some unique capabilities. However, the percentage of what’s new to what’s familiar is, on average, small.

Figure 1.1 illustrates some similarities between Pascal, Fortran and C².

Capability	Pascal	Fortran	C
assignment	<code>a := b</code>	<code>a = b</code>	<code>a = b</code>
conditional	<code>if (a = b)</code>	<code>if (a .eq. b)</code>	<code>if (a == b)</code>
repeat loop	<code>for i:=1 to 10 do</code>	<code>do 100 i=1,10</code>	<code>for(i=1;i<=10;i++)</code>
file open	<code>reset/rewrite</code>	<code>open</code>	<code>fopen/open</code>
file include	<code>{ \$I file }</code>	system-specific	<code>#include <file></code>
data types	integer, real, char	integer, single, character	int, float, char
array	<code>day: array [1..31] of integer</code>	<code>integer day(31)</code>	<code>int day[31]</code>
constants	<code>const pi = 3.14</code>	<code>parameter (pi = 3.14)</code>	<code>#define pi 3.14</code>
comment	<code>(* . . . *)</code> or <code>{ . . . }</code>	<code>"C"</code> or <code>"c"</code> in column 1	<code>/* . . . */</code>

Figure 1.1: Language Capability Comparison

3 C Language Capabilities

In this section we discuss the capabilities of the C language. Many of these capabilities will be familiar to Fortran users, while other capabilities are analogous to assembler languages.

3.1 C Compiler Pre-Processor

While the pre-processor is not part of the language but part of the compiler, statements that are actually directives to the pre-processor are. As the name suggests, the pre-processor reads in the program, strips out comments, uses the directives found in the program to include header files, expand macro definitions and substitute actual values for symbolic constants, then passes the results to the compiler to translate from human-readable form to machine-readable form.

3.2 Derived Data Types Built From Built-In Data Types

C allows the construction of derived data types. These are called records in Pascal and are called structures in C. The new type is built from, or derived from, the data types built into the language. For example, a vector in 3-dimensional space,

$$\vec{x} = (x_1, x_2, x_3)$$

²The table refers to UCSD Pascal, Fortran 77 and ANSI C.

can be built in C as follows:

```
struct vector {
    double  x1;
    double  x2;
    double  x3;
};
```

where “double” is a C built-in (primitive) floating-point data type. We can then define position and velocity vectors variables,

```
struct vector position, velocity;
```

We can also assign a type name to structures. This is called “defining a type name,” or “typedef” for short, and looks like

```
typedef struct vector {
    double  x1;
    double  x2;
    double  x3;
} VECTOR;
```

Our definition of position and velocity vectors now looks like,

```
VECTOR position, velocity;
```

Much more complex data types can be built, using any previously-defined derived types and C built-in types.

3.3 Assignment Statements

An assignment statement is a statement of the type

$$\textit{something} = \textit{something else};$$

and is the only non-control or non-declaration statement in the C language. All other C statements perform some type of declaring, defining, including, branching, looping or testing.

3.4 Conditional Execution

Conditional execution statements allow for controlling what code gets executed based on a set of conditions. Referring to the spherical coordinate calculation program in §4.2, one type of conditional execution statement is an **if** statement. If the radius (r) is not zero, then calculate the coordinates, else set all spherical coordinates to zero. Another type of conditional execution statement is a **switch** statement, analogous to a **case** statement in Pascal, and operates similar to an **if**.

3.5 Repetitive Execution

Repetitive statements control program flow by repeating execution of a block of statements until some condition is satisfied. These statements are **for**, **while** and **do . . . while**. A **for** performs the same function as the **do** statement in Fortran. Both **while** and **do . . . while** repeat a block of code while something is either true or false.

3.6 Recursive Execution

Recursive execution is a very powerful technique for writing simple, fast algorithms. An algorithm is a series of statements that perform a specific, logical task, such as converting cartesian to spherical coordinates or finding the eigenvalues of a matrix. Recursive execution means that a routine calls itself. A classic example is a subroutine to calculate the factorial value for an integer. If we have an integer n , then “ n factorial” is written $n!$, and has the value

$$n! = \prod_{i=1}^n i = n(n-1)(n-2)(n-3) \dots (2)(1)$$

If the value passed to our subroutine (n) is greater than 1 then multiply n by the value returned by another (recursive) call to the subroutine, passing that call the value $n - 1$. This looks like,

```
double factorial( int n )
{
    if ( n > 1 )
        return( n * factorial( n-1 ) );
    else
        return( 1 );
}
```

For $n = 5$ this subroutine would call itself recursively 4 times.

3.7 External Definition of Subroutines

Subroutines referenced from the **main()** routine need not be defined in the same physical file that contains the **main()**. They do, however, require declaration so that the compiler knows what they are, what their arguments are and what data type they return (if they return anything at all). In the spherical coordinates program of §4.2 the functions **acos()** and **sin()** are defined in the C math library *libm.a*. We included *math.h* since this is where the declarations of C library math functions are. It is instructive to examine some of the C include files provided to support the libraries. They can usually be found in the directory */usr/include* on most Unix systems.

3.8 Address and Bit Operations

C provides the capability to perform address arithmetic and manipulate bit values. To most users these are the most confusing concepts. These capabilities are used frequently, however, and add a dimension to C that is lacking in other languages such as Fortran or Pascal. Manipulating addresses and bits will be discussed in the next two chapters.

3.9 Formatted and Unformatted Input/Output

As surprising as this sounds, there are NO built-in C language input or output language statements. All I/O is performed by subroutines provided in the C libraries. While at first this may appear to be a serious weakness it is in fact a non-issue. All C releases provide a standard library of routines for both formatted (text) and unformatted (binary) input and output. Figure 1.2 lists some of the common routines used. Note that an entire chapter is devoted to just this library of routines. Also note in the table that formatted file routines can also input data from and write data to the console.

4 Basic Structure

C programs are a collection of routines with associated definitions for constants, variables, and other such objects. A C program must have a routine designated as the starting point. In Fortran and Pascal the starting routine is the routine which begins with a **program** statement. In C the starting point is a routine called **main**. Other routines contained in or referred to by the program are **subroutines** to the **main** routine.

Type	Console	Formatted File	Unformatted File
open		fopen	open
input	scanf gets getchar	fread fscanf fgets	read
re-position		fseek rewind	lseek
output	printf puts putchar	fwrite fprintf fputs	write
close		fclose	close

Figure 1.2: Sample C Input/Output Routines

4.1 Program Components

A typical C program will contain the following building blocks:

- **Comments.** Comments are essential to program readability and maintenance. A well-documented program is commented throughout to assist in understanding the code.
- **Include statements.** C programs are usually broken into many separate files. Common definitions and declarations are grouped together into what are referred to as **include**, or **header**, files. For example, the necessary definitions and declarations for using standard C input-output (I/O) library routines are found in the file *stdio.h* (standard **i/o** header file). Include statements are an example of a type of C compiler pre-processor directive. The pre-processor will be discussed later.
- **Subroutine declarations.** Subroutines must be declared as to what data type they are (int, double, char, etc.). For example, the sine trigonometric function is declared in the file *math.h* and looks something like

```
double sin(double x)
```

which says that sine takes a double-precision argument x and returns a double-precision value. A subroutine declaration may also include what the argument types are. This is usually a good idea since this allows the compiler to check for consistent usage. All subroutines are assumed to be of some default type by the compiler (usually integer), unless a subroutine declaration is found, before it is used in the program.

- **Symbolic constants.** C allows a user to define a name to be associated with something. For example, if the maximum length of a directory name is 30 characters, then defining `MAX_DIR_LENGTH` to be 30 allows the program code to be both more readable and easier to maintain. C language convention is to use all upper-case letters in the names of symbolic constants. Fortran 77 uses `PARAMETER` statements to define constants. Declaring symbolic constants is another C pre-processor directive.
- **main().** This routine is required in all C programs, and defines the starting point of the program.
- **Variable definition.** These statements allocate memory for the information processed by the program.
- **Executable statements.** These statements do the information processing, such as assignment, conditional processing, recursive calculation and input/output. C statements can start in any column and are terminated by a semi-colon. Since statements end with “;” we can add whitespace and write statements spanning several lines. The extra whitespace and newline characters are ignored by the compiler.
- **Subroutine definitions.** A subroutine is just that—a subroutine. Subroutines are usually declared before usage, but can be defined anywhere outside the main routine. Subroutines cannot be defined inside another routine. To understand what the difference is between *declaring* and *defining* we illustrate with the sine function. The declaration tells the compiler what type the function is and (optionally) what the arguments are, whereas the definition is the executable code that “does” the computing. In our example of the sine function the definition might look like,

```
/* This routine will return the trig. sine function,
   in radians, for the passed-in value. */

double sin( double x )
{
    double    y;

    y = x - pow(x,3.0)/6.0 + pow(x,5.0)/120.0 -
          pow(x,7.0)/7040.0 ...
    /* taylor series expansion for sin(x) */
    return(y);
}
```

where `pow(a,b)` returns a^b .

4.2 Structuring Programs using Components

Figure 1.3 presents the layout of a program built from the components listed above. When starting to develop a program the “components” need not be actual C language code. The idea is that we first develop the overall structure of the program in plain language statements describing what the program is to do, then expand each statement and continue expanding until all ideas are broken down into basic statements which correspond to language statements. We then write the necessary C code to perform each described task. Often the plain language statements are kept in the subroutine or main routine in the form of comments, which then serve as documentation. This is sometimes referred to as top-down development, and is but one of many software development methodologies.

As a simple example, if we need to write a quick program to convert cartesian coordinates to spherical coordinates, we might start with the plain language statements

```
initialization and set-up

define main routine
read in values from user
convert to spherical coordinates, checking
    for all possible conditions
print results
exit
```

then expand this to

```
/* This program reads in 3 cartesian coordinates
   from the user and translates them to spherical
   coordinates.  */

/* set-ups */

#include <stdio.h>
#include <math.h>

/* define main routine */
int main( int argc, char *argv[] )
{
    /* set-up */
    double x, y, z;
    double r, phi, theta;
```

```
/* read in values from user */
printf( "Enter 3 coord. (spaces between): " );
scanf( "%lf %lf %lf", &x, &y, &z );

/* convert to spherical coordinates, checking
   for all possible conditions */
r = sqrt( x*x + y*y + z*z );
if ( r > 0.0 )
{
    theta = acos( z / r );
    if ( ( theta > 0.0 ) && ( theta < M_PI ) )
    {
        if ( fabs(x / ( r * sin(theta) ) ) >= 1.0 )
            phi = 0.0;
        else
            phi = acos( x / ( r * sin(theta) ) );

        if ( y < 0.0 )
            phi = (2.0 * M_PI) - phi;
    }
    else
        phi = 0.0;
}
else
{
    r = 0.0;
    theta = 0.0;
    phi = 0.0;
}

/* print results */
printf( "r = %lf, phi = %lf, theta = %lf\n",
        r, phi, theta );

return(0); /* exit */
}
```

This program works (at least on my system). A good exercise is to type this into a file and build an executable program (assuming the name *yourfile*) using the command `make yourfile LDFLAGS=-lm`, then run the program by typing *yourfile*. The program will prompt you then display the results. Note that no error checking is done on the input beyond checking *r*.

5 Summary

In this chapter we introduced the building blocks of a C program and discussed basic capabilities. In the next several chapters we will refine the capabilities by detailing how one constructs the language syntax for these building blocks.

Main Program File:	External Subroutines File:
<pre> /* comments */ #include <header files> #define constants #define macros declare subroutines; declare global variables; /* comments */ type main(int argc, char *argv[]) { declare local variables; executable statements; call user-written subroutines; call library subroutines; ... return(some status value); } /* comments */ type subroutine(arguments) { declare local variables; executable statements; call user-written subroutines; call library subroutines; return(results); } </pre>	<pre> /* comments */ #include <header files> #define constants #define macros declare subroutines; /* comments */ type subroutine(arguments) { declare local variables; executable statements; call user-written subroutines; call library subroutines; return(results); } /* comments */ type subroutine(arguments) { declare local variables; executable statements; return; } </pre> <p data-bbox="727 1234 1122 1417">External file of subroutines supplied with the C compiler. All releases of C have these libraries. They include math, I/O, string, memory management and other functions.</p>

Figure 1.3: Program Built from Components

Chapter 2

DATA TYPES

6 Introduction

In this chapter we discuss the C language capability to implicitly allocate memory for data storage.¹ Statements that define variables for use look something like,

```
double    inertial_velocity;  
int       loop_counter;
```

These statements instruct the C compiler to allocate a specific amount of memory (the amount depends on the data type) and to associate all references to the variable name with the location of the allocated memory. The definition of a variable implies the allocation of memory to store the value of the variable, hence implicit allocation of memory. When we assign a value to `inertial_velocity` or `loop_counter` we are actually directing that a value be written to computer memory. The memory location is called an *address*, in direct analogy with the concept of the addresses used by the Postal Service to deliver mail. The idea is the same—it is a very important concept. Every variable we define is allocated storage, the location of which is the variable’s address in memory.

We first digress from the various data types and their characteristics to first fill in some terminology used when describing computer memory.

7 Bits and Bytes

Again we call upon the reader to step back from the (infinite) details while we attempt to fill in some terminology. We will be discussing bits, bytes, words, blocks and other ideas. These terms describe different quantities of memory.

The fundamental memory element is the bit. A bit can be assigned either a 1 or 0 value. With only 2 values available, computers process data not in base 10 but base 2. Computer

¹C provides the capability to explicitly allocate a specified chunk of memory (`malloc()` and related functions). These will be covered in the chapter on the C standard libraries.

memory represents the 0 or 1 value, in hardware, as some type of “on” or “off” condition, similar to a light switch—it’s either on (1) or off (0). Incidentally this is what is meant when computer on/off switches are instead labeled 1/0.

Next in line is the byte, which is (usually) composed of 8 bits. Character data is typically stored one character per byte. With eight bits, and two values (0 and 1) per bit, this allows a byte to be assigned 2^8 or 256 different values, ranging from 0 to 255.

A word of computer memory is composed of either 2 or 4 bytes, depending on the hardware. Also dependent on the hardware and operating system configuration is a block of data, typically 512 or 1024 bytes. Data read in or written to external storage (disks, tapes, etc.) are usually written in block increments.

The internal representation of floating-point numbers are often discussed in terms of “single-precision” or “double-precision.” The single and double reference is to either 4 or 8 bytes. Double-precision numbers have more bits to represent the number and therefore provide more numerical accuracy. For some floating-point calculations, such as balancing our check-books, the quantities involved don’t push the accuracy limits of the single-precision representation and therefore are sufficient. Other calculations, such as a surface of section mapping in non-linear dynamics requiring many iterations, may become sensitive to very small round-off/truncation errors and therefore justify employing double-precision data types.

Figure 2.1 summarizes several different memory reference terms and their characteristics². The machine-specific value limits for any particular data type can be found in the files *limits.h*, *float.h* and *math.h*.

Type	Size (bits)	Value Range (unsigned)	# of Values
bit	1	$0 \leftrightarrow 1$	$2^1 = 2$
nybble	4	$0 \leftrightarrow 15$	$2^4 = 16$
byte	8	$0 \leftrightarrow 255$	$2^8 = 256$
integer	16	$0 \leftrightarrow 65535$	$2^{16} = 65536$
word	32	$0 \leftrightarrow 4294967295$	$2^{32} = 4294967296$
single-precision	32	$\sim 10^{-45} \leftrightarrow 10^{38}$...
double-precision	64	$\sim 10^{-324} \leftrightarrow 10^{308}$...

Figure 2.1: Memory Terms and Sizes

²These values are hardware dependent. The values listed are typical for workstations. We will discuss signed and unsigned storage in the following sections.

8 Built-In Types

There are four built-in data types in C, which we list in Figure 2.2. We discuss each in sections below.

<i>char</i>	⇒	character (text) data
<i>int</i>	⇒	integer data
<i>float</i>	⇒	lower-precision floating-point data
<i>double</i>	⇒	higher-precision floating-point data

Figure 2.2: C Built-In Data Types

8.1 Character Data

Character data is information that we typically associate with characters typed from a keyboard. Names, messages, page headings, numbers represented by character digits (e.g., -14 is actually, as you read it, a string of character digits) and newline (carriage return) are examples of characters or strings of characters. The amount of memory allocated to store one character is implementation-dependent, but is typically 1 byte (8 bits).

8.2 Integer Data

Integer data has no fractional part and are analogous to whole numbers. The range of values an integer may take depends on the machine implementation. Integers are usually either 2 or 4 bytes in length. The value ranges for integers are dependent on the qualifiers used in the definition, as discussed in the section on data type qualifiers below.

8.3 Floating Point Data

Floating-point numbers are analogous to real numbers and are stored in mantissa-exponent form. The range of accuracy for a floating-point number is determined by the size of the mantissa in bits and the size of a floating-point number is determined by the number of bits used to represent the exponent. The table showing value ranges for the different types reflects the fact that there are (usually) more bits for the exponent of a double than for the exponent of a float.

8.4 Void Data Type and its Uses

The void type is used in a subroutine definition and declaration to indicate that the subroutine returns no data or that the subroutine takes no arguments. For example,

```
void    error_msg_and_abort( int error_number )
{
    ...
    return;
}
```

could be the definition of a routine that handles fatal errors in a program. Defining the type of the subroutine to be void indicates that the routine will not return a value. All routines not defined to be void will return a value. If the routine had a return statement with a value,

```
...
return( something );
}
```

then the compiler would produce an error message, since the routine is of type void and therefore *cannot* return a value.

Defining a routine such as,

```
int    get_processor_status( void )
{
    ...
    return( results );
}
```

indicates that the subroutine takes no arguments. If the routine is called somewhere and arguments are specified then the compiler will print an error message for that line. The void argument list explicitly says that there are *no* arguments.

A variable can be defined as a void type if it is a pointer. We will cover pointers and addressing later in the chapter.

9 Scope of a Variable

If a variable is defined inside a subroutine then the variable is referred to as being *local* to that routine. Variables defined outside a routine are called *global* variables, and are said to

have global scope. In other words, any subroutine has access to global variables but only the defining subroutine has access to local variables.

Global variables are allocated memory and initialized before any program statements are executed. Globals are allocated memory only once and remain allocated for the duration of program execution. Hence a global variable maintains whatever value is assigned throughout the time the program runs. When the program exits all memory allocated for program and data storage is released by the operating system³.

A local variable is allocated memory every time a subroutine is called and is released (memory freed) upon subroutine exit. Therefore the value of a local is not maintained between subroutine calls. This can be modified by using the *static* qualifier, discussed below. The allocation and freeing of non-static local variables is automatic when the subroutine is called and therefore locals are referred to as *automatic* variables. Local variables are either automatic or static.

A variable may also be defined inside a block of code, enclosed by braces ({ and }). The scope of the (local) variable is then the block of code. If a variable is defined locally and also exists as a global variable then the local definition is used and access to the global variable is therefore lost.

10 Data Type Qualifiers

There are several qualifiers that may be used with the data type name to change either the scope or interpretation of the variable.

Unsigned. The unsigned qualifier is used to specify that character or integer values stored in the variable be treated as unsigned (i.e., positive) quantities. Defining a variable using the unsigned qualifier looks like

```
unsigned int loop_counter;
```

Usually the most-significant bit is used as a sign bit, allowing both positive and negative values. Using the unsigned qualifier suppresses this usage and all bits are used to represent the value being stored. Figure 2.3 illustrates a few examples. Note that the ranges are machine-dependent and are listed only for illustration.

Short. The short qualifier is used to define the smallest version of an integer. Since short implies short int the int may be omitted. Depending on the hardware the size of a short (and long, discussed below) may or may not be smaller than the unqualified int

³There can be a difference between a program *exiting* and a program being *interrupted*. This possible difference is computer-specific.

Type	Signed Range	Unsigned Range
char	-128 ↔ 127	0 ↔ 255
int	-32768 ↔ 32767	0 ↔ 65535
long	-2147483648 ↔ 2147483647	0 ↔ 4294967295

Figure 2.3: Signed/Unsigned Value Ranges

type. A general rule is that if you know the data being stored can be handled by a 2-byte integer then define the variable as short or int. If the data range is unclear but may exceed the limits of a 2-byte integer then use long.

Long. The long qualifier is used to define a larger version of the data type, and is the counterpart to short. Using long by itself implies asking for a long int. Long may be used with int and double data types. A long data type is defined to be larger (in bytes) than or equal to the type being qualified. For example, if the hardware uses 4-byte integers as a default then a long int will also (usually) be 4-byte integers and they will be the same size. However, if an int is only 2 bytes then a long int will still be 4 bytes and the long will be larger than an int.

Extern. Storage for a variable is allocated when a variable is defined. The extern qualifier on a variable declares the variable but indicates that the variable is defined external to the current subroutine. For example, in a subroutine the statement

```
unsigned int    ssme_status;
```

indicates that `ssme_status` is local to the subroutine, while

```
extern unsigned int    ssme_status;
```

states (declares) that this subroutine will be using the variable `ssme_status` but that it's defined somewhere else. Without the extern qualifier the statement would be a definition and the compiler would allocate storage for the variable.

Static. Adding the static qualifier to a local variable indicates that the value stored in the variable is maintained from one subroutine call to the next. Variables that are not defined as static are allocated memory when the subroutine is called and the memory is freed when the subroutine is exited. A useful example is if a subroutine wishes to keep a count of how many times it has been called,

```
static int    execution_counter;
```

With the static qualifier the value of `execution_counter` is maintained from call to call. We could then increment `execution_counter` at the start of the subroutine to track the number of calls to the routine. Without the static qualifier the value of `execution_counter` would always be lost when the subroutine exits.

The static qualifier may also be used with a function definition/declaration. In this context the static qualifier serves to limit the scope of the subroutine to the file its defined in. We will discuss below that a function name, devoid of the parenthesis (which together constitute the subroutine operator, discussed in Chapter 3), is a pointer to the subroutine. In this context the subroutine name is just like a variable, and variables have a scope. “Scope” is a concise way of describing how widely known a variable name is. Variables defined in a subroutine are local to that subroutine, so their scope is limited to the subroutine. Variables defined as static in a file of subroutines, but outside any of the routines, are known to all the subroutines of that file. The variable’s scope is said to be the entire file of subroutines. If a variable is defined outside any subroutine with no static qualifier then it is global in scope, since any subroutine defined in any file can access the variable (all routines know the variable exists—they have access to it).

Register. The register qualifier indicates that the variable will be used often and to therefore allocate a hardware storage register for the variable, if possible. Using hardware registers for frequently used variables speeds the execution of a routine. The register qualifier is a *request* for hardware register use—if the computer cannot allocate a register then the register request is ignored. Note that using the register qualifier restricts the variable to an automatic variable and that we cannot determine the address of a register variable.

11 Derived Types

A derived data type is a type built from previously-defined types. We can build elementary types from the built-in C types and build more complex types from our elementary types. For example, a state vector consists of a position vector, a velocity vector and a time. C allows the definition of a new type (vector) and the definition of a `state_vector` type built from the vector type and double for the time.

11.1 Structures

A structure is a collection of variable definitions. Structures are built by combining variables of previously-defined data types together. A structure definition has the form

```
struct name {
    variable definition statement;
    variable definition statement;
    variable definition statement;
    ...
};
```

where *name* is the name for the structure and the variable definition statements are the usual statements like,

```
double    time;
int       loop_counter;
```

Our example of the state vector would be set up as follows.

```
struct vector {
    double    x;
    double    y;
    double    z;
};

struct state_vector {
    struct vector    position;
    struct vector    velocity;
    double           time;
};
```

The first structure defines what a vector is composed of. We then use the definition in the `state_vector` structure. Position and velocity are variables of type `struct vector`. We used `struct vector` just as we would use `double`, `int`, `long` or any other data type.

The elements in a structure are referred to as *members*. To reference a member of a structure concatenate the structure variable (e.g., `position` or `velocity`) together with the element name (e.g., `x`, `y` or `z`) using either of the *structure member operators* “.” or “->”. For example, a statement to set the `y`-member of the `velocity` would look like

```
velocity.y = 14654.2;
```

The “->” is the operator used if the structure variable is a pointer. Pointers are discussed later in the chapter.

We can build complex structures by building up definitions. To develop our example further, suppose we defined a trajectory structure to contain information which described a spacecraft trajectory. We would use previous structures in the new structure,

```
struct trajectory {
    int             vehicle_number;
    struct state_vector  state;
    double          altitude;
    ...
}

struct trajectory    shuttle_traj;

shuttle_traj.state.position.z = 0.0;
...
```

The last line illustrates how to trace down through the structures to get to members.

11.2 Defining New Type Names (typedef)

While it is true that structures are used as new types, we must include “struct” every time we wish to define a variable of that type. What a *typedef* does for us is allow us to assign the structure definition a *type name*. We then use that name when we define variables. To illustrate we repeat the last example but use typedefs.

```
typedef struct vector {
    double    x;
    double    y;
    double    z;
} VECTOR;

typedef struct state_vector {
    VECTOR    position;
    VECTOR    velocity;
    double    time;
} STATE_VECTOR;

typedef struct trajectory {
    int             vehicle_number;
    STATE_VECTOR    state;
    double          altitude;
    ...
} TRAJECTORY;

TRAJECTORY    shuttle_traj;
```

```
shuttle_traj.state.position.z = 0.0;
...
```

Examining the `state_vector` structure we see that the syntax of the variable definitions looks the same. There are two data type names used, namely `VECTOR` and `double`.

Typedefs are not restricted to use with structure definitions. Any data type can be renamed to a new data type. A natural question is “Is this useful?” In fact this is very useful. Suppose, for example, a package of software were developed which assumed that integer values processed would never exceed the range of allowed values for the data type `int`. Now suppose later that the volume of data processed became so large that integer overflow started to occur. If all variables were defined to be type `int`, then they would *all* require changing to type `long`, which can handle the larger values. If, on the other hand, the variables were defined as type `Package_INT`, typedef’ed as

```
typedef int    Package_INT;
```

then only one line of code would require a change,

```
typedef long   Package_INT;
```

This is an example of using the capabilities of C to produce flexible, portable, easily-maintained software.

11.3 Unions

A union is a definition whereby two or more variables share the same physical memory, and is similar to the Fortran equivalence statement. When several variables are unioned together writing a value to any one changes the value for all, since they share the same memory. A small example will illustrate the occasional usefulness of unions.

Suppose we wished to define a two-dimensional vector. We would use a shortened version of the earlier `VECTOR` type. However, suppose we wished to extend this definition to handle polar coordinates as well as cartesian? We would still have two elements, but now they would be a radius and an angle. We could use the `x` and `y` members of our previous `VECTOR` type but this would be misleading—the `y` name implies a `y` axis and hence a cartesian space. We can extend our definition using unions as follows.

```
typedef struct vector {
    enum dataType {CARTESIAN,POLAR};
    /* enumerations are covered below */
```

```
    union coordinate1 {
        double      x, r;
    } c1;
    union coordinate2 {
        double      y, theta;
    } c2;
} VECTOR;

VECTOR    position;

position.dataType = POLAR;
position.c1.r = 155.0;
position.c2.theta = M_PI/5.0;
```

The members `x` and `r` occupy the same memory address. Access to the members of a union uses the same syntax as for structures. Notice in our example that we have added an extra variable to help keep track of the vector's coordinate system.

Unions are not restricted to variables of the same type. The amount of memory allocated when the union of differing types is defined will be the memory required to store the largest type.

11.4 Enumeration

An enumeration is a definition that states that a variable can take only certain values. These values are set up when the enumeration is defined and consist of a list of symbolic constants. The values assigned to the symbolic constants depend on how we set up the enumeration.

When we define a variable to be of type enumeration the compiler will check the usage to insure we assign the variable a value only from the list of symbolic constants. The value assigned to an enumeration variable cannot be calculated⁴. Enumerations are used when we know a variable will be assigned certain values and no others. For example, an engine is either on or off, so this suggests defining an engine status variable as

```
enum engine_status { ON, OFF };
```

The list of symbolic constants start with the value 0 and are incremented for each new constant, so the symbol `ON` has a value 0 and `OFF` has a value 1. This can be changed by setting the constant to a value. Two examples will illustrate.

```
enum ascent_major_mode { PRE_LAUNCH = 101, STAGE1,
```

⁴As usual, this depends on the compiler.

```
        STAGE2, OMS1, OMS2, COAST };
```

```
enum ops = { ASCENT_NOMINAL=1, ASCENT_RTLS=6,  
            DESCENT=3, ORBIT=2};
```

If we attempt to assign `ops` a value other than one of the constants in the list, the compiler will issue a warning message. Also, using enumerations adds to the readability of a program since tests of an enumeration's value are comparisons with the symbolic constants, such as,

```
if ( ascent_major_mode == STAGE1 )  
{  
    ...  
}
```

11.5 Arrays

Arrays are sets of variables of some type referenced by a single name and one or more subscripts. C allows arrays to be defined for any type, including structures, pointers and enumerations. If we wish, for example, to define an array of 10 position vectors (the vector type was defined in the section on structures and typedefs) we would write,

```
VECTOR    position[10];
```

The specifier “[10]” indicates that the array has one dimension and 10 elements. A three dimensional array with 10, 5 and 3 elements respectively would have a specifier “[10][5][3].”

C assigns the first element per dimension in an array the subscript 0. Fortran and Pascal start subscripts at 1. This is a very important point. The subscripts for our array of position vectors ranges from 0 to 9, NOT from 1 to 10. The reason C works this way will be discussed when we introduce pointers. For an array defined as,

```
int    engine_status[MAX_ENGINES][MAX_SUBSYSTEMS];
```

the subscripts range from 0 to (MAX_ENGINES - 1) and from 0 to (MAX_SUBSYSTEMS - 1). A common mistake when working with C software is to overwrite the program/data memory lying just beyond the end of an array by exceeding the subscript range of an array (e.g., writing data to a `position[10]` element, which doesn't exist).

12 Pointers

C uses the pointer types to support indirect addressing. A pointer type is a variable which is assigned an address value instead of the usual integer, floating-point or other value. When we reference ordinary variable names in programs what we are actually using is an address in memory. Assigning a variable a value is actually a *store value at address* instruction.

The three operators used with pointer and address manipulation are *, & and ->. The & operator is used to determine the address of a variable. The * operator indicates that the variable being operated on is a pointer. The -> operator associates a pointer to a structure with a structure member.

As with most sections of this chapter, and especially so in this section, we illustrate the usage of pointers with examples. A short example would be,

```
int main( int argc, char *argv[] )
{
    int    i, *j;    /* define an integer (i) and */
                  /* a pointer to an integer (*j) */

    i = 4;          /* assign a value to i */
    j = &i;         /* assign to j the address of i */

    *j = 194;       /* change the value pointed to by */
                  /* j, which is i in this case */

    ...
}
```

Since `j` points to `i` the statement `*j = 194;` actually changes `i`. That we can change one variable (`i`) by changing another (`*j`) is one example of indirect addressing. We set `i` indirectly, through a reference to `*j`, which points to `i`.

A key point to understand is that `j` is **not** an integer variable, rather, it is a pointer to an integer. Pointers are not of type `int` or `char` or `float` or `double` or any other type, they are *pointers*. We can ask questions about definitions such as,

```
double    *vehicle_mass;
```

such as what is the data type of `vehicle_mass`? Of `*vehicle_mass`? A useful technique is to draw a box around the object in question—what's left over is the type. For example, if we draw

```
double    *vehicle_mass;
```

```
double *vehicle_mass;
```

then we see that `*vehicle_mass` is of type `double` while `vehicle_mass` is a pointer to a `double`.

The definition above for `*vehicle_mass` is a pointer to a `double`. What does `vehicle_mass` point to? An uninitialized pointer is one of the hardest and most subtle errors in C programming to find and fix. An uninitialized pointer points to an unknown address in memory. The initial value of a pointer is compiler-dependent and (usually) cannot be predicted. Using an uninitialized pointer is like playing Russian Roulette with memory—the address being used may be part of the program being run or (worse yet) part of the operating system.

12.1 Call by Value and Call by Reference

Pointers are often used when passing arguments to subroutines. Arguments passed to a subroutine are usually passed by value, which means that the value of the argument is duplicated in the subroutine. A call to the `sin()` function such as

```
y = sin(x);
```

causes the value of `x` to be passed to `sin()`, not the location in memory where `x` is stored. If we wish to pass to a subroutine the location of the variable, and not just the variable's value, then we need to pass the *address* of the variable. This is known as a call by reference. Suppose we wrote a subroutine to calculate the sin of an angle and to change the angle from degrees to radians if the angle is degrees. We would then be required to both return a sin value and a changed angle value, both from one subroutine. An easy way to do this is with pointers.

```
x = 135.0;                /* set x to 135 degrees, which */
                          /* is (3 PI)/4 radians      */
y = MySin( &x );         /* call our subroutine    */
printf( "Sin(%lf) = %lf\n", /* print the results, showing */
        y, x );          /* y = 0.707 and x = (3 PI)/4 */
```

...

```
double MySin(double *angle) /* our sin function def.    */
{
    if ( *angle > 2.0*M_PI ) /* test if angle is greater */
    {                        /* than 2 PI, scale if it is */
```

```

        *angle = *angle * (M_PI / 180.0);
    }

    return( sin( *angle ) ); /* return sin value      */
}

```

12.2 Array Names as Pointers

If we define an array of position vectors, as before,

```
VECTOR    position[10];
```

this directs that memory be allocated for 10 copies of the VECTOR type. The memory is allocated in one piece, so that `position[3]` is stored immediately after `position[2]`. In C the array name without subscript notation (i.e., the `[]`) is defined to be a pointer to the type of the array, in our case a pointer to a VECTOR type. Therefore the array name points to the first member of the array.

When in C an array element is accessed the actual address of the element is determined by multiplying the size (in bytes) of the array type by the subscript and adding the beginning address for the array (i.e., the array name). This is why arrays start with element 0 instead of 1. Every array element reference causes this address calculation to be done⁵. If when looping through the array we instead keep track of the address then we save processor time. For our array of vectors, this might look something like,

```

VECTOR    position[10];
VECTOR    *curVector;
int        i;

curVector = position;      /* start address      */

for ( i = 0; i < 10; i++ ) /* note we loop from 0 to 9 */
{
    curVector->x = 0.0;     /* initialize each member to 0 */
    curVector->y = 0.0;     /* note the use of "->" since */
    curVector->z = 0.0;     /* curVector is a pointer      */

    curVector = curVector + sizeof( VECTOR );
                          /* add to the address the size */
}

```

⁵Some compilers optimize this process to avoid duplicate address calculations.

```

        /* in bytes of the VECTOR type */
    }

```

What we save in using `curVector` is the multiplication of the size of the `VECTOR` type by the subscript. We just add the size of the `VECTOR` type to our pointer and it then points to the next array element.

If we were writing a large subroutine we could also use this technique to save processor time. If, for example, we were referencing `position[5]` frequently in our large routine then each reference causes an address calculation. If we calculate the address once,

```

    curVector = &(amp; position[5] ); /* calculate address once */

```

then we can reference the members (`x`, for example) using `curVector->x` directly and avoid the address calculation each usage of `position[5].x` would require.

There are many areas where pointer usage facilitates structured program design or optimizes processor throughput.

12.3 Arrays of Pointers

While this sounds like an excessive complication, arrays of pointers are occasionally useful. One use is when passing command line arguments from the operating system to a program. Recall an earlier `main()` definition,

```

int    main( int argc, char *argv[] )
{
    ...
}

```

The `char *argv[]` argument can also be written `char **argv`. The `argv` argument is an array of pointers. Each pointer points to one of the command line arguments. The `argc` argument tells the program how many arguments there are. Recall our use of boxes around parts of a definition. We employ them here to illustrate `argv`. The description indicates the type of what's boxed.

```

char  *argv[]  < is of type char
char  *argv[]  < is a pointer to char
char  *argv[] < is an array of char pointers

```

A quick program to echo the arguments passed to the program is as follows, and illustrates using an array of pointers. Note that in this example we loop from 0 up to and including `argc`. Argument number 0 is the name of the program being run and the first argument to the program is pointed to by `argv[1]`. There are therefore `(argc + 1)` elements in the `*argv[]` array.

```
#include    <stdio.h>

int  main( int  argc, char *argv[] )
{
    int    i;                /* loop counter          */

    for ( i = 0; i < argc; i++ )
    {
        printf( "Argument %d is %s\n", i, argv[i] );
    }
}
```

The `printf` format indicator `%s` requires a pointer to a string, which is what `argv[i]` is. A string is an array of characters with the last character being a null (`'\0'`) byte.

12.4 Void Pointer Type

A void pointer is an untyped pointer. Until now we have been discussing pointers to something, such as a char pointer. A void pointer points to an address in memory but no type is associated with the address. In a previous example we defined `curVector`, which was a pointer to a `VECTOR`. When we use memory allocation routines we may use the memory for anything we wish. We may store double precision numbers there or integers or characters or anything. The type of pointer returned by memory allocation routines is the void type. In the next chapter we will discuss the `typecast` operator, which takes one data type and casts it to another. When we allocate memory we cast the returned generic (i.e., void) pointer to whatever type we wish. This `typecast` usually reflects how we intend to use the memory.

Void pointers are also used in subroutines when the type of pointer being passed in can vary. Again, this just says that void pointers are the most basic type of pointer (generic). A classic example is a subroutine that moves blocks of memory. Whether the blocks are arrays of doubles or character arrays or whatever, this routine simply moves a specified number of bytes from one starting address to another. Since this is a generic memory transfer routine the pointer arguments for this routine would be void.

12.5 Pointers to Functions

Just as variables are allocated storage in memory, and therefore have an address, subroutines also are assigned a starting address. Although this may appear rather abstract, pointers to subroutines are used frequently. A project worked on by the author several years ago serves as an excellent example.

Suppose we were to modify a compact disc player to interface with a computer. The front panel pushbuttons would be disabled and instead the computer would send the equivalent electronic signals to activate the button hardware. Software to control the player would then need subroutines to perform the various button “pushes.” Also, we would need to program everything—not just button pushes but button releases after an appropriate time (we used 1/2 second as the “push” interval), for if the button were released too fast the hardware wouldn’t recognize the “push” event. One major part of the problem then was to fashion an event queue to hold what buttons we wanted pushed and when we wanted the button released. The following code could serve as a basis for this queue. We only set up two buttons for illustration.

```
#define    MAX_QSIZE    100

int  process_event(  int    (*)( void *, double),
                    /* pointer to int routine */
                    void*, /* generic data ptr.   */
                    double ); /* time to activate */

int  play_button( void *, double ); /* push play routine */
int  stop_button( void *, double ); /* push stop routine */

typedef  playdata {          /* structure to hold      */
    ...                    /* play button data      */
} PLAYDATA;

typedef  stopdata {         /* structure to hold      */
    ...                    /* stop button data      */
} STOPDATA;

typedef  eventdata {       /* this structure holds   */
    int    (*routine)();   /* the event queue data  */
    void   *routine_data; /* for a single event    */
    double time;
} EVENTDATA;
```

```
PLAYDATA   play_data;           /* define variables      */
STOPDATA   stop_data;
EVENTDATA  event_queue[ MAX_QSIZE ];
           /* allocate the event queue */

event_queue[0].routine = play_button;
           /* event 0 is play_button */
event_queue[0].routine_data = &play_data;
event_queue[0].time = current_time + 10.5;
           /* occur 10.5 seconds later */
           /* than current time      */

event_queue[1].routine = pause_button;
           /* event 1 is stop_button */
event_queue[1].routine_data = &stop_data;
event_queue[1].time = event_queue[0].time + 5.0;
           /* occur 5 seconds after  */
           /* the play started       */

curEvent = 0;
number_of_events = 2;
exit_from_loop = FALSE;

while ( !exit_from_loop )
{
    if ( event_queue[ curEvent ].time >= current_time )
    {
        /* call the routine if at */
        /* the indicated time     */
        (*event_queue[ curEvent ].routine)(
            event_queue[ curEvent ].routine_data,
            event_queue[ curEvent ].time );
    }
    ...
}
...
```

Just as an array name alone is a pointer to the start of the array, a function name alone is a pointer the start of the routine. This can be seen in the initialization of `event_queue[0]` and `event_queue[1]`. The syntax for calling the routine is illustrated in the `if` statement at the end of the code.

As can be seen from the while loop the event queue structure must handle calling whatever routine we need and pass data appropriate for the routine. What we would set up in the play routine, for example, is that at the end of the subroutine we would queue a “play button release” event for 1/2 second after the button push and de-queue the push play event. The main routine would loop through the event queue and dispatch any events that were queued to be performed at the current time. The main loop would continue to cycle through the events until an external program queued an exit event.

When the project using this software was implemented the author then wrote a crude language to allow programming a series of button events. The “programs” in this language were called test sequences and were easy for an end-user to type in. Such a test sequence might look like,

```
Spin up disc
Fast-forward to time index 2:00:00
Play for 10 seconds
Fast-forward to time index 2:56:70
Play for 5 seconds
Stop disc
Open drive door
```

This software was used to test compact discs during manufacturing as part of the quality assurance process.

13 Rules for Defining Variables

A variable must be defined before it is used. Traditionally variables are defined at the start of a routine (between the opening brace “{” for the routine and the first executable statement). Variable names must be unique for the first 31 characters and cannot begin with a numeric digit. Names are case sensitive and can also include the underscore (–) character. The reserved words in C, such as `if` and `while`, cannot also be used as variable names.

It should not be surprising that we cannot define a variable with the name “4.” If this were allowed then we could write statements like

```
4 = 5;
```

Also, how could we initialize a variable to the literal value 4, since 4 is now a variable?

We list some variable definition examples for illustration.

<code>inertial_velocity</code>	<	uses the underscore
<code>InertialVelocity</code>	<	alternate naming idea
<code>a2i</code>	<	cryptic but legal
<code>_ExampleOfVar</code>	<	underscore can be leading char.
<code>4Example</code>	<	illegal—digit cannot be 1 st char.
<code>Inertial_Velocity</code>	<	NOT the same as <code>inertial_velocity</code>

14 Constants

Up to now we have been describing data types as applied to defining variables for data storage. We can also use data constants. These are specified as literal values and cannot change. We discuss below the various constant types and their representation.

14.1 Character Constants

A character constant is any literal value that can fit in one character of memory storage. Characters are represented as numeric values in a computer, and the correspondence to an alphabetic upper- or lower-case letter, numeric digit, punctuation or other character is dependent on the character code table used by the computer. There are two primary code tables, EBCDIC⁶ and ASCII⁷. An ASCII code table can be found in Appendix B.

Character constants are represented by the character surrounded by single quotes. Character constants can also be represented by the corresponding hexadecimal or octal value. Special characters such as carriage return or tab have built-in special sequences, which we list in Figure 2.4[6]. We illustrate character constants with a few examples.

```
char a, b, c, d, e, f; /* declare some variables      */
a = 'X';              /* character surrounded by quotes */
b = 0x58;             /* an X in hexadecimal           */
c = '\n';             /* a special character (newline) */
d = '\130';          /* an X in octal                 */
e = '\0';             /* a null character (=0)         */
f = '\'';            /* set f to the quote character  */
```

⁶Extended Binary Coded Decimal Interchange Code

⁷American Standard Code for Information Interchange

Description	Char	Sequence	Description	Char	Sequence
newline	NL	<code>\n</code>	backslash	<code>\</code>	<code>\\</code>
Horizontal tab	HT	<code>\t</code>	question mark	<code>?</code>	<code>\?</code>
vertical tab	VT	<code>\v</code>	single quote	<code>'</code>	<code>\'</code>
backspace	BS	<code>\b</code>	double quote	<code>"</code>	<code>\"</code>
carriage return	CR	<code>\r</code>	octal number	000	<code>\000</code>
formfeed	FF	<code>\f</code>	hex number	hh	<code>\xhh</code>
audible alert	BEL	<code>\a</code>			

Figure 2.4: Special Character Sequences

14.2 String Constants

A string is a sequence of characters terminated by a null character. String constants are specified as a sequence of characters surrounded by double quotes. The string functions in the C libraries assume null-terminated strings. When strings are typed in a program, the compiler will add the null character, so character arrays must be sized to account for this extra character.

We again illustrate with an example.

```
char    msg[20];          /* declare variable          */
strcpy( msg, "This is my string" );
                          /* copy a constant string to msg[] */
                          /* the null char. is in msg[17]   */
printf( "Another string constant\n" );
                          /* string constant printed to the */
                          /* console                          */
```

14.3 Integer Constants

An integer constant is a sequence of digits, optionally preceded by a minus sign, and optionally followed by a qualifying character or two. The suffix characters are “L” or “l,” and “U” or “u.” The “u” indicates an unsigned integer and the “l” indicates a long. Integer constants may be specified in either octal, decimal or hexadecimal. Examples of integer constants include,

```
int     i, j, k;         /* declare variables          */
```

```

unsigned   v;
long      large_val;
i = 0x23ff;          /* set i to 9215 decimal using hex   */
j = -2042;          /* set j to a negative value;       */
k = 021777;        /* set k to 9215 decimal using octal */
v = 70000u;        /* set v to an unsigned value       */
large_val = -1245123L; /* either l or L, just as either u  */
                    /* or U, can be used                */

```

14.4 Floating-Point Constants

A floating-point constant is formed in a similar way to integer constants and can take the general form,

$$\{\pm\}nnn.mmm\{e|E\{\pm\}ppp\}\{f|F|l|L\}$$

where “nnn,” “mmm” and “ppp” are integer values. Unsigned floating-point constants are taken to be positive. Terms surrounded by “{ }” are optional and the vertical bar indicates the different terms that can be used. Therefore the exponential form is optional but the decimal point is not. When forming the exponent either of “e” or “E” may be used but one is required. Floating-point constants are by default of type `double`. Using “f” or “F” indicates the constant is of type `float` and the “l” or “L” indicates the constant is of type `long double`. A few examples will illustrate the syntax.

```

double     x, y, z; /* declare variables           */
long double ld;
float      a, b;
x = 110.44253;     /* basic form of a constant       */
y = 9.11e-31;      /* exponential notation            */
z = -12.39E-12;    /* exponential notation with neg.  */
                  /* exponent and upper-case E      */
ld = -34.22e81;    /* long double                     */
a = -3.141592654F; /* constant of type float using F  */
b = 9103226.0f;    /* another float constant using f  */

```

15 Logical Values and Boolean Values

By convention a “logical” value is either a true or false value, with false taking the numerical value 0 and true any non-zero value. Note that true is any non-zero value. Testing the results

of a logical comparison for a specific numerical true value is usually disastrous. A logical comparison may return 0 for false and 1 for true on one computer, but may correctly return 0 for false and -1 for true on another computer. Testing for true by comparison to 1 would work on one system but not another.

While it may appear unnecessarily complicated, the proper tests for true or false are “either false or (not>false.” If something is “not false” then it (obviously) must be true. What this formulation does for us is to base true/false on only the value of false, namely 0. Anything not false (0) is true.

In contrast to the true/false logical values, the boolean values true/false are 0 for false and 1 for true. While this is a definition, in reality we use boolean tests just as we use logical tests—either false or not false. The subtleties between these will be discussed when we cover logical and boolean operators in the next chapter.

Chapter 3

C OPERATORS

16 Introduction

C programs define objects such as variables and subroutines. The C operators provide the ability to manipulate variables and subroutines to perform the task at hand. When we calculate a value we are using arithmetic and assignment operators. When we test a value or condition we are using relational (logical or boolean) operators. The idea of using operators is so basic that we use them even when we don't realize we're using them. A classic example is how we write negative numbers. If we wish to express the value 3.14159 as a negative number we write -3.14159 . The “ $-$ ” is an operator—the negation operator.

The concepts behind the majority of the C operators are derived from mathematics and machine-level (assembler) programming. The statement

$$a = b + c;$$

is understood to add **a** and **b** and assign the result to **c**. This is the usual ordering of an algebraic statement in that the left side of the equal sign is the result (we would never write $4 = x$, for example). While many are familiar with arithmetic and algebra far fewer are familiar with assembler programming. A common assembler operation (op code) is an increment instruction, usually abbreviated “inc.” There is a C operator $++$ which does the same thing—add one to an operand. Another example is the indirect addressing operator $*$, which says to use the value of the operand as a pointer to the final address. These operations are common in machine-level programming but not in 3rd generation languages such as Pascal and Fortran.

The discussion in this chapter of the C operators groups the operators by what they operate on. The section on special operators is for operators which don't fit easily anywhere else. We begin with the operators most familiar to and proceed to those that are a bit more unusual.

Operator	Description	Type	Association	Precedence
-	Negation	unary	Right-to-Left	14
++	Increment	unary	Right-to-Left	14
--	Decrement	unary	Right-to-Left	14
*	Multiply	binary	Left-to-Right	13
/	Divide	binary	Left-to-Right	13
%	Remainder (modulo)	binary	Left-to-Right	13
+	Add	binary	Left-to-Right	12
-	Subtract	binary	Left-to-Right	12
* =	Shorthand multiply/assign	binary	Right-to-Left	2
/ =	Shorthand divide/assign	binary	Right-to-Left	2
% =	Shorthand modulo/assign	binary	Right-to-Left	2
+ =	Shorthand add/assign	binary	Right-to-Left	2
- =	Shorthand subtract/assign	binary	Right-to-Left	2

Figure 3.1: Arithmetic Operators

17 Arithmetic Operators

The arithmetic operators are listed in Figure 3.1. The operators +, - (subtract), - (negation), * and / are familiar and perform the expected arithmetic computations. The negation operator takes an operand and changes it's sign.

The shorthand operators can be easily understood by demonstrating their use as follows.

```
double    position_x = 12.4;    /* declare two variables    */
int       major_mode = 103;

position_x = position_x * 4.2; /* these two statements    */
position_x *= 4.2;            /* perform the same multiply */

major_mode = major_mode + 2;  /* these two statements also */
major_mode += 2;              /* perform the same addition */
```

The shorthand operators work by taking the variable on the left of the assignment operator (=) and using it as the first operand on the right.

The divide operator will return an integer value (only) if the operands are both integers. If we divide 5 by 3 the answer is 1. The remainder operator (%) is the counterpart to

the divide operator for integers in that $5 \% 3$ is 2, or the remainder of a divide operation. The remainder operator is also known as the modulo operator since the operation being performed is actually a modulo operation.

The last two arithmetic operators are the increment ($++$) and decrement ($--$) operators. These operators act on variables only, whereas the other operators such as $+$ may operate on variables or numbers. Also, the $++$ and $--$ operators may be either prefixed or postfix to a variable. Prefixing one of these operators (e.g., $++i$) will increment the variable before it is used, while postfixing the operator (e.g., $i++$) will first use the current value of the variable then increment. Therefore the three statements

```
i = 4;
j = i++;
k = ++i;
```

will set j to 4 then increment i , then increment i to 6 and assign the value to k . Therefore $i = 4$, $j = 4$ and $k = 6$. This ability to either prefix or postfix the operator can cause subtle program execution errors. For example,

```
char    string1[] = "This is an example string";
char    string2[40] = {'\0'}; /* set up the string var.'s */
char    *strPtr1, *strPtr2; /* define 2 pointers to be */
                                /* used in a string copy */
strPtr1 = string1; /* ptr to start of 1st string */
strPtr2 = string2; /* ptr to start of 2nd string */
while ( *strPtr1 ) /* copy (or so we think) the */
    *strPtr2++ = *strPtr1++; /* test string */
```

This section of code will miss the “T” in the word “This” because the pointer is incremented first before we assign $*strPtr2$ the first character. The while loop should actually look like,

```
while ( *strPtr1 ) /* correct copy of the test */
    *strPtr2++ = *strPtr1++; /* string */
```

This example serves to illustrate the importance in understanding how operators work and what can happen when we don’t pay attention to details.

18 Logical Operators

We list these operators in Figure 3.2. The arithmetic comparison operators take as their argument some type of number. We cannot, therefore, compare character strings with these

Operator	Description	Type	Association	Precedence
<	Arithmetic less than	binary	Left-to-Right	10
<=	Arithmetic less than or equal	binary	Left-to-Right	10
>	Arithmetic greater than	binary	Left-to-Right	10
>=	Arithmetic greater than or equal	binary	Left-to-Right	10
&&	Logical and	binary	Left-to-Right	5
	Logical or	binary	Left-to-Right	4

Figure 3.2: Logical Operators

operators (there is a library routine to do this, however). The comparison operators work as our intuition tells us they should. The results of the operation is a logical true or false.

The logical “and” and logical “or” operators take as their operands two logical values. “And” results in a logical true value if both operands are true, and false otherwise. The logical “or” results in a true value if either operand is true, and false otherwise.

19 Boolean Operators

Operator	Description	Type	Association	Precedence
!	Boolean not	unary	Right-to-Left	14
==	Equal to	binary	Left-to-Right	9
!=	Not equal to	binary	Left-to-Right	9

Figure 3.3: Boolean Operators

These operators, listed in Figure 3.3, work similarly to the logical operators, but return a boolean true or false. If the operand to the “not” operator is true, then the operation results in false, and if the operand is false then the results is true. The equal and not equal operators compare numeric value and return either (boolean) true or false, and work as we expect them to. If two values are equal, then the == operation results in true.

One important point is worth mentioning. We must take care when testing two floating-point values for equality. The equality test checks for **exactly** equal. A simple example will illustrate the problem. Take a “scientific” calculator and perform the following two computations.

- Calculate 13 to the power 0.5 by using the y^x function.
- Calculate the square root of 13 by using the $\sqrt{\quad}$ key.
- Subtract the two answers.

Where we would expect a zero answer to the subtraction, most calculators will display an answer something like 1^{-13} . While this is a small number, it is not zero, so the results of the two calculations would fail an equality test despite the fact that they are, in fact, equal. The problem lies in the algorithms used to perform the numerical functions. These algorithms are actually finite approximations using series expansions.

20 Bitwise Operators

Operator	Description	Type	Association	Precedence
\sim	One's complement	unary	Right-to-Left	14
\ll	Left shift of bits	binary	Left-to-Right	11
\gg	Right shift of bits	binary	Left-to-Right	11
$\&$	Bitwise and	binary	Left-to-Right	8
$ $	Bitwise or	binary	Left-to-Right	7
\wedge	Bitwise exclusive or	binary	Left-to-Right	6
$\ll=$	Shorthand bit shift left/assign	binary	Right-to-Left	2
$\gg=$	Shorthand bit shift right/assign	binary	Right-to-Left	2
$\&=$	Shorthand bitwise and/assign	binary	Right-to-Left	2
$\wedge=$	Shorthand bitwise excl. or/assign	binary	Right-to-Left	2
$ =$	Shorthand bitwise or/assign	binary	Right-to-Left	2

Figure 3.4: Bitwise Operators

The bitwise operators are listed in Figure 3.4. The shorthand forms of the bitwise operators work just like the shorthand arithmetic operators. The \sim operator changes the value of the operand by switching the operand's bits. If a bit is 1 it is changed to 0, and 0 bits are changed to 1. If we have a two-byte integer value of 0111001010010111 then the one's complement of this value is 1000110101110100. The \sim operator only operates on integer values. We can operate on character values with \sim if we first typecast the value to an integer.

The \gg and \ll operators shift bits right or left. The usage is $i \ll n$, where n is an integer number which specifies the number of positions to shift. For example, with the 1-byte character value $i = 10010111$ the operation $i \ll 2$ results in 01011100. All bits

have been shifted left two positions. The left shift always fills the positions vacated with zeros. The right shift operator will do the same with operands that are of the type unsigned, but right shifting a signed operand will fill the vacated positions with the sign bit on some computers but fill with 0 on others. The left-most bit in a signed integer is used to indicate the sign of the value, 0 for positive and 1 for negative.

The effect of left or right shifting is to multiply or divide the operand by 2^n , where again n is the number of positions shifted. To illustrate this with an example. If $i = 00000001$ in bits (the upper 8 bits of the 2-byte integer are zeros and are not listed), which is the value 1, then $i = i \ll 2$ results in $i = 00000100$, which is the value 4 or $1 * 2^2$. The subsequent operation $i = i \ll 3$ results in $i = 00100000$, which is the value 32 or $4 * 2^3$. If we then write $i = i \gg 1$, and i is an unsigned int, then i is $i = 00010000$ or 16 (i.e., $32/2^1$).

The last three operators take their two operands and perform bit-by-bit comparisons. Their operation is easily understood if we focus on what happens to each bit. The `&` operator results in 1 *per position* if both bits in the position are 1, and 0 otherwise. In other words, the result is 1 if the first operand bit is 1 and the second operand bit is 1. This is why the `&` operator is called the bitwise “and” operator. The `|` operator results in 1 if either the first operand bit is 1 or the second operand bit is 1, and is called the bitwise “or” operator. The `^` operator is called a bitwise exclusive or operator. The result of `^` per bit is 1 if the bits differ (i.e., one of them is 1 and the other 0), but will result in 0 if both bits are 0 or 1.

A usage of these bitwise operators can be illustrated with an example of a computer controlling the operation of a set of lights in a building. The computer can control whether the lights are on or off. We can define a `light_status` integer and use the bits to represent the status of a light. If we had 9 lights in our building, then perhaps overnight we would want every third light on, but also we wish to sequence the lights so that all parts of the building are lit periodically.

We start by setting `light_status = 0x0049`, which is in bits 0000000001001001. In other words, light 1, 4 and 7 lights (and bits) are on. After an hour we set `light_status` as follows,

```
light_status = ( (light_status << 1) & 0x01ff ) |
               ( (light_status & 0x0100) >> 9);
```

The first part of this shifts the bits up by 1 position and sets to 0 all bits but the low 9. The second part gets the 9th bit and results in either 0 or 1. If the 9th bit is 1 (i.e., the 9th light is on) then light 3, 6 and 9 are on and we next want lights 1, 4 and 7 lights on. The second part sets the first bit to the previous value of the 9th bit. The lights and bits sequence is illustrated in Figure 3.5.

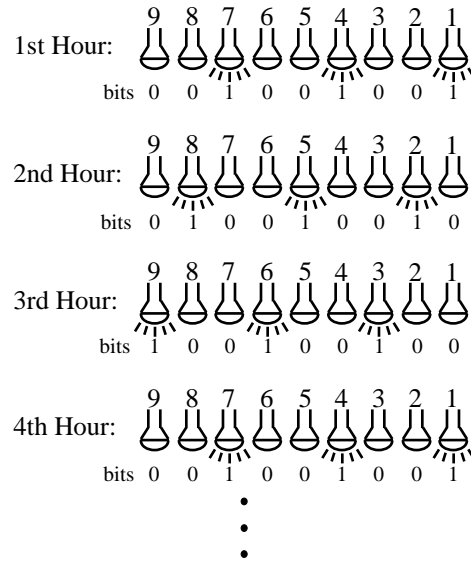


Figure 3.5: Sequence of Lights Example

21 Address Operators

Operator	Description	Type	Association	Precedence
->	Struct/Union member using ptr.	binary	Left-to-Right	15
&	Address of	unary	Right-to-Left	14
*	Indirect addressing	unary	Right-to-Left	14

Figure 3.6: Address Operators

The address operators are collected in Figure 3.6. The structure/member operator (->), where the structure variable is a pointer, has already been discussed and is repeated here only as a reminder that its usage does involve a pointer. The two remaining operators we have discussed in Chapter 2. The * operator causes the the value of the operand to be used as the address of where the actual operand is located. The & operator results in the address of the operand instead of the operand's value. We review these concepts in an example.

```
int    i, j, *k;    /* declare some variables. i and j are */
```

```

                                /* integers, and k is a ptr. to an int. */
i = 32113;                       /* assignment in decimal      */
j = 0xf0ce;                       /* assignment in hexadecimal   */
k = i;                             /* k now is 32113, which if interpreted */
                                /* as an address is probably an error */
k = &j;                             /* k now points to j          */

printf( "k's value points to %x\n", *k; );
                                /* the value printed is 0xf0ce (in hex) */
                                /* since k points to j. The value of k */
                                /* is the address of j, but the * oper- */
                                /* ator changes the interpretation of k.*/

```

22 Special Operators

The special operators either act on operands or change the meaning of an operand. The function call operator `()` is an example. A function name, by itself, is a pointer to the function. Adding the `()` suffix changes the interpretation from a pointer to the function to an actual call to the function. Any arguments to the function are listed between the parenthesis, separated by commas (another operator). The array index operator `[]` works similarly to the `()` operator.

The structure member operators `.` and `->` were discussed in chapter 2 and associate structure names with their constituent members. The `.` operator is used if the structure variable is not a pointer, and `->` is used if it is.

Speaking of the comma operator, it serves to separate a list of things. Usage in a routine call is familiar, but the following usage is somewhat more unusual.

```

int    i, j, k;                   /* declare some variables */
i = (j = 4, k = 6);

```

The results of the assignment statement is to set $i = 6$, $j = 4$ and $k = 6$. Another usage is in the initialization part of a `for` statement, which we'll cover in the next chapter.

The *sizeof* operator does what the name implies—it returns the size of its operand in bytes. The operand can be either a data type or a variable. The usage (syntax) is as follows.

- `sizeof variable`
- `sizeof(type name)`

Enclose the operand of *sizeof* in parenthesis if it is a type and skip the parenthesis if the operand is a variable. *sizeof* will return the total size of whatever is being operated on. Therefore, the size of an array is the size of a single array member (i.e., the number of bytes for the variable's type) multiplied by the number of elements in the array. The size of a structure is the sum of the sizes of the members.

The typecast operation takes a value of one type and converts it to another type. If we wish to store the whole-number part of a floating-point number in an integer variable, we typecast the floating-point value to an integer as follows.

```
i = (int)(3.14159);
```

If we wish to use one type of pointer as another type, we typecast it to the desired type. The *typecast* operator is the correct way to convert from one type to another.

The last special operator, *?:*, is actually a shorthand form of an *if* statement. We illustrate with an example.

```
int    i, j, k;           /* declare some variables    */
i = (j == 21 ? -1 : 1);  /* if j is 21, then assign i */
                          /* the value -1, else assign 1 */
```

The result of the *?:* operator depends on the conditional test. If the conditional test results in true (boolean or logical) then the value following the question mark is the resulting value of the operator. If the test is false then the value following the colon is the result. This operator can be used to check values before usage, correcting them if the value is outside some legal bounds. For example,

```
double  x, y;           /* declare some variables    */
y = -4.5;               /* an illegal value for sqrt() */

x = (y >= 0.0 ? sqrt(y) : sqrt(-y) );
                          /* if y is positive, then    */
                          /* assign x the sqrt of y,    */
                          /* assign x the sqrt of -y    */
```

The special operators are illustrated in Figure 3.7.

23 Operator Precedence

If two operators are used in the same line then which one is performed first? The answer to this question is why we assign operators an evaluation order and precedence. The operators

Operator	Description	Type	Association	Precedence
()	Function call	unary	Left-to-Right	15
[]	Array element	unary	Left-to-Right	15
.	Struct/Union member	binary	Left-to-Right	15
->	Struct/Union member using ptr.	binary	Left-to-Right	15
(<i>type</i>)	Typecast	unary	Right-to-Left	14
<i>sizeof</i>	Size of operand in bytes	unary	Right-to-Left	14
?:	Conditional	binary	Right-to-Left	3
,	Multiple expressions separator	binary	Left-to-Right	1

Figure 3.7: Special Operators

in a C statement are evaluated in either right-to-left or left-to-right, and some are evaluated before others. In our tables of operators we have ranked the operators so that the higher the precedence operators are evaluated first, with operators of equal precedence evaluated in their association order. Appendix C summarizes the operators in one table.

Chapter 4

C LANGUAGE STATEMENTS

24 Introduction

We present in this chapter the statements used to carry out the task our programs are to perform. These statements differ from data definition and declaration statements in that data statements allocate storage but are not “executed.” Data definition and declaration statements are directives that specify what memory storage we plan on using, the data type (interpretation) of the storage and how the data storage is organized. The statements that we discuss here are those that operate on the data storage and control the flow of programs.

25 Statement Format

The preceding chapters have demonstrated the basic statement format, but we present it here for completeness. Statements are either simple, compound or null statements.

25.1 Simple Statements

A simple statement is any single C statement terminated by a semi-colon. All of the following are simple statements.

```
i = 243;
strcpy( char_ptr, "My String" );
j = i < 200 ? -100 : 100;
y = sin( x );
```

White space is ignored in a C statement. We can therefore place a C statement in any column and put more than one statement on a line. For the purpose of readability C statements are usually placed one per line and indented uniformly.

25.2 Compound Statements

We build compound statements from simple ones by enclosing a group of simple statements in braces “{ }.” The compound statement is not terminated by a semi-colon. A compound statement may be used wherever a simple statement is allowed. Compound statements may be nested (one contained within another). We can therefore group simple or compound statements into larger compound statements.

What use can we make of compound statements? Their use is already familiar. An example is in the `for` statement, which we discuss in a later section. The syntax of a `for` statement is

```
for ( init. condition; while condition; end-of-loop stmt. )
    C statement
```

and an example of this is

```
for ( i = 0; i < 10; i++ )
{
    j *= ( i - 4 );
    printf( "Current j value is %d\n", j );
}
```

Notice that the example C statement is a compound statement. If we delete the braces then the `for` loop would only execute the `j` assignment 10 times, since this would then be the next statement (simple in this case). The control flow statements we discuss later in the chapter control the execution of a C statement, whether simple, compound or null.

25.3 Null Statement

A null statement consists of only a semi-colon. We can use the null statement wherever a C statement is required. Null statements are typically employed when whatever we wish to do is done inside a conditional test. A classic example is when copying strings using pointers, we code the actual copy as follows.

```
char    *str_A, *str_B;
...
while ( *str_A++ = *str_B++ )
    ;
```

The string copy executes until the assignment returns 0 (i.e., the null character terminating the string). Since the copy is done inside the `while`, there is nothing to be done by the C

statement, and we therefore use the null statement.

26 Assignment Statement

We mention the assignment statement here only to include it in the list of different types of statements. We have seen assignment statements in most examples in the previous chapters. The assignment statement uses the assignment operator = and is the only non-control flow statement in C. All other executable statements either perform a test or transfer control to another part of the program.

27 Statement Format Style

Although we can place more than one C statement on a line, this is generally not done. We discuss here some suggested guidelines to formatting C text to enhance readability.

C statements are usually placed one per line and separated with enough white space to facilitate reading. Code bunched together takes less lines on the screen or paper but is harder to read. We never notice the use of white space in a newspaper, but it's there. If a statement is longer than about 80 characters then break the line at a logical place and indent the remaining text. The statements that make up a compound statement should be indented to the right of the braces enclosing the compound statement.

Comments are usually welcome documentation but frivolous comments just clutter the code. For example, the statement `i++` increments `i` so don't add a comment to the right that says the program is incrementing `i`. If `i` is an index counter then say 'increment index counter.'

The goal is to make the code easier to understand. These suggestions to enhance readability are illustrated in the coordinate conversion program listed in §4.2.

28 Calling Subroutines

A "call" to a subroutine occurs when we transfer control to the subroutine. Control passes to the routine when we refer to the name of the routine with the function call operator, `()`. In other words, a subroutine is called when we use it in a C statement. We have seen many examples. The following C statement calls the subroutine `sqrt()` to calculate the square root of the argument.

```
y = sqrt(x);          /* calculate the sqrt */
```

This line “calls” the `sqrt()` subroutine. The name `sqrt` is a pointer to the function, but is used as a subroutine call when we operate on the name with the function call operator “`()`.” We can use a subroutine call that returns a value as an operand, illustrated with a rather exaggerated example.

```
y = cos(x) * sin( log( sqrt(x) ) );
```

The argument to the `log()` function is the result of the call to `sqrt()`. Note that if a subroutine is type `void` then we cannot use it as an operand since there isn’t anything to operate on.

29 Control Flow Statements

We use control flow statements to change what is done depending on some condition encountered during program execution. If something is false then do C statement *A* else do statement *B*. While we haven’t reached the end of a file read the next line from the file and process it. Ideas such as these are what we use control flow statements to implement.

The *test* condition we refer to below is test returning either true or false. For example, `i == 4` would be either true or false depending on *i*. Refer to the sections in Chapter 3 for the logical and boolean operators that return a true or false result. Note that this test condition is a C statement, returning a logical or boolean (i.e., true or false) value.

29.1 If

An `if` statement is constructed as follows.

```
if ( test is true )
    C statement for true condition
else
    C statement for false condition
```

The C statements may be any of the three types (simple, compound or null). The `else` part of the `if` may be omitted. We list several examples. The first shows two `if` statements each with an `else` clause. The first *if* also has another `if` nested inside the “C statement for true condition.” The “C statement for false condition” for the first `if` is another `if`.

```
if ( i == 1 || i == 2 || i == 4 )
{
    printf( "Value of i is positive and valid\n" );
```

```
    if ( i == 4 )
        j = 3;
    else
        j = 2;
}
else if ( i < 0 )
{
    printf( "Value of i is negative (and valid)\n" );
    j = 1;
}
else
{
    printf( "Value of i (%d) is invalid\n", i );
}
```

Notice where the braces are placed and the indentation. This allows for easier program readability. The second “false condition” is a simple C statement (a `printf()` call) but is surrounded by braces. This is not required for a simple statement but facilitates adding additional statements to the `if` later.

Another example is,

```
if ( val >= 0.0 ) {
    return( sqrt( val ) );
}
else {
    return( sqrt( -val ) );
}
```

and is an alternate (longer) form of the return statement illustrated in §30.3 using the `?:` operator. Notice the format of the braces. This is another common coding style.

Further examples of `if` statements can be found in the previous chapters (e.g., §4.2).

29.2 Switch

The `switch` statement is a concise form of an `if` and is used when the condition tests are for a series of cases. For example, a `switch` statement is often used to process command line arguments. The general form of a `switch` is

```
switch( value )
{
```

```
    case constant1:
        C statement 1;
        break;

    case constant2:
        C statement 2;
        break;

    default:
        default C statement ;
        break;
}
```

This is equivalent to the if statement,

```
if ( value == constant1 )
{
    C statement 1;
}
else if ( value == constant2 )
{
    C statement 2;
}
else
{
    default C statement;
}
```

We may set up as many cases as we wish. The `constant` values are just that—constants. They must be either integer or character constants. The `break` statement is discussed in §30.1 and causes the `switch` statement to be exited. If we omit the `break` then all the statements after the matched `case` are executed, including those for subsequent cases, until either the end of the `switch` or a `break` is encountered.

If we had a program that accepted three options (`i`, `r` and `d` for example) from the command line then we would code a command line argument processor loop as follows.

```
for ( i = 1; i < argc; i++ )
{
    /* loop for all arguments */
    switch( argv[i][0] )
    {
        case 'r':
```

```
        ...
        break;

    case 'd':
        ...
        break;

    case 'i':
        ...
        break;

    default:
        printf( "Invalid argument (%s)\n",
               argv[i] );
        exit(0);
        break;
}
}
```

If we added an option `a` that did the same thing as `r` then the switch would be changed to look like,

```
...
case 'a':
case 'r':
    ...
    break;
```

Since there is no `break` after the `case 'a':` the program flow would continue with whatever `case 'r':` does, which is what we wanted.

The `default` case, if present, is always matched. Therefore if no previous cases are matched (or if we forget to include the `break` statement) before the `default` case then whatever is specified under the `default` is executed. If we place the `default` case first in the list of cases it would always be done. Usually the `default` case handles some exception condition and is placed after all other cases.

29.3 While

A `while` loop executes a C statement while a condition is true. The `while` condition is tested before each execution. The general form is

```
while ( condition )
    C statement
```

and an example to print out a countdown, using a system-specific delay function, would look like,

```
i = 10;
while ( i )
{
    printf( "Countdown...%2d\n", i-- );
    system( "sleep 1" ); /* delay 1 second */
}
printf( "Liftoff.\n" );
```

The test here is a logical one, checking for a value not zero and therefore true. The loop exits and “Liftoff.” is printed when *i* reaches 0. Note that “Countdown... 0” is not printed since the condition test is performed before the C statement (compound in this case) is executed.

Another example would be to loop to read in a line from a file and process it so long as no error occurs on the read command. Such a loop would look something like,

```
#define MAXLENGTH 80

FILE    *ourFile;          /* set up the variables */
char    ourLine[ MAXLENGTH ];
...
while ( fgets( ourLine, MAXLENGTH, ourFile ) != (char *)0 )
{
    ...process the line of text read in...
}
```

The function `fgets()` reads the next line of text from a file and returns a pointer to the start of the text read in. If an error occurs on the read (e.g., end-of-file) then `fgets()` returns a null character pointer value, which is what we test for. “While not a null pointer process the line read in” is a statement describing the loop.

29.4 Do..While

The `do..while` is similar to a `while` loop except that the test is performed at the end of the loop. The general form is

```
do
    C statement
while ( condition );
```

and our countdown example changes to

```
i = 10;
do {
    printf( "Countdown...%2d\n", i-- );
    system( "sleep 1" ); /* delay 1 second */
}
while ( i );
printf( "Liftoff.\n" );
```

It is important to understand that the loop will be executed once *always*. If the test condition is false the loop will still execute once before the test is performed and the loop exited.

29.5 For

The `for` statement is used to repeat a C statement and is also similar to a `while`. In fact, we can always code a `for` loop as a `while` loop. which will be shown below. The general form of a `for` is

```
for ( Part1; Part2; Part3 )
    C statement
```

`Part1`, `Part2` and `Part3` are all C statements. `Part1` is an initialization and usually sets up a loop counter. `Part2` is a conditional test and is checked at the start of each iteration of the loop. `Part3` is a statement executed at the end of each loop. Note that we may omit any or all parts of a `for`. We can therefore write

```
for ( ; ; )
    C statement
```

if we wished and could use the `break` statement to exit the loop when so desired. The `for` statement is intended for use as a counter-driven looping statement. Putting complex constructions in `Part1`, `Part2` or `Part3` is strongly discouraged.

An example of a `for` loop is copying a string and translating it to all upper-case letters.

```
for ( i = 0; i <= strlen( MyString ); i++ )
```

```
{
    MyNewString[i] = toupper( MyString[i] );
}
```

The function `strlen()` returns the length of a string. Notice that this loop also copies the null character terminating the string. In this example `Part1` is `i = 0`, `Part2` is `i <= strlen(MyString)` and `Part3` is `i++`.

As stated earlier, we can set up a `while` loop to do the same thing a `for` would do, and looks like for our example above,

```
i = 0;
while ( i <= strlen( MyString ) )
{
    MyNewString[i] = toupper( MyString[i] );
    i++;
}
```

Refer back to the descriptions of `Part1`, `Part1` and `Part3` to compare their descriptions with where they are placed in the equivalent `while` loop.

30 Control Flow Modifiers

Control flow statements execute statements conditionally. What is executed usually depends on some condition. The `if` statement is a classic example. The operation of the control flow statements can be modified with the C statements `break`, `continue` and `return`. We use modifiers to handle an exception or to alter the flow of a loop. If we wished to print a list of the first 15 Fibonacci numbers, but *not* the 9th one, then we would use a `continue` statement to skip to number 10 when we reached the 9th number.

30.1 Break

A `break` statement will cause the program to skip to the first C statement following the current `switch`, `for`, `while` or `do..while` statement. If the program is performing a `while` loop, for example, a `break` will exit the loop and continue at the first statement following the `while`. We have already seen the `break` in the `switch` statement above.

30.2 Continue

Where the `switch` exits the current loop, the `continue` statement causes the program to pass control to the `Part3` statement of a `for`, and the condition test for the `while` or `do..while`. For a `while` loop this occurs at the beginning of the loop and for a `do..while` this occurs at the end. Recall that the `Part3` statement of a `for` is performed as the last step before checking the `Part2` condition test at the start of the `for`. To code our Fibonacci example we cited above (assuming we've already written a subroutine `fibonacci()` to calculate a Fibonacci number),

```
for ( i = 0; i < 15; i++ )
{
    if ( i == 9 )
        continue;

    printf( "Fibonacci number %d is %d\n",
           i, fibonacci(i) );
}
```

As a note on readability, compare this with

```
for ( i = 0; i < 15; i++ ) {
    if ( i == 9 ) continue;
    printf( "Fibonacci number %d is %d\n",
           i, fibonacci(i) ); }
```

30.3 Return

The `return` statement passes program control from a subroutine back to the calling routine. `return` may also pass back a value to the caller. If a routine has no `return` statement then program control passes back to the caller after the last C statement in the subroutine has been executed.

A `return` can be placed anywhere in a routine. This includes the `main()` routine and is usually used there to pass back a status value to the operating system.

A `return` passing a value back to the caller looks like

```
...
return value;          /* one form          */
return( value );      /* an equivalent form */
...
```

We omit *value* when no value is to be passed back. Using the `return` statement in the code of a subroutine rather than at just the end relieves the burden of keeping track of what to return and also of trying to avoid executing code if (for example) an error condition exists. If we write a subroutine to calculate both the square root and natural log of a number, then a negative or zero value passed in is an error and we return immediately with an error value,

```
int    calc_math( double  val,      /* incoming # */
                double  *sqrt_val, /* sqrt() of val */
                double  *log_val ) /* log() of val */
{
    if ( val <= 0.0 )
    {
        return( -1 );      /* indicate error with -1 */
    }

    *sqrt_val = sqrt( val );
    *log_val  = log( val );

    return( 0 );          /* no error, so return 0 */
}
```

A routine to calculate just the square root regardless of what is passed in might look like,

```
double  calc_sqrt( double  val ) /* incoming # */
{
    return( val >= 0.0 ? sqrt( val ) : sqrt( -val ) );
    /* return the sqrt() of the absolute */
    /* value of the incoming number    */
}
```

Chapter 5

C INPUT/OUTPUT LIBRARY

31 Introduction

32 Files and Devices

33 Character I/O

34 Formatted (Text) I/O

35 Unformatted (Binary) I/O

Chapter 6

C STANDARD LIBRARIES

- 36 Introduction
- 37 String Functions
- 38 Math Functions
- 39 Memory Management Functions
- 40 Data Conversion Functions
- 41 Date/Time Functions
- 42 System and Program Exit Functions
- 43 Character Conversion and Testing Functions
- 44 Sorting and Searching Functions
- 45 Miscellaneous Functions

Appendix A

C PRE-PROCESSOR

The pre-processor is a program that reads in a C source code file to be compiled and performs certain operations such as stripping out comments, including header files, expanding macro definitions and substituting actual values for symbolic constants. The results of this pre-processing is then passed to the compiler to translate from human-readable form to machine-readable form.

Pre-processor directives are statements that begin with a pound sign (#). Usually these statements begin in column 1 but can be start in any column. Figure A.1 lists the C pre-processor directives.

Directive	Description
#define	Define constant or macro
#undef	Remove definition (un-define)
#include	Include a file
#if	If ... else test
#ifdef	If symbol defined
#ifndef	If symbol not defined
#line	Change source line number and filename for error diagnostics in compilation
#error	Write error diagnostic message
#pragma	Computer-specific command directive

Figure A.1: Pre-Processor Directives

A.1 Macro Definition

A macro is very similar to a subroutine except that wherever the macro is used the code for the macro is substituted by the pre-processor instead of a subroutine call being made. Yes, this sounds confusing—a few examples will help to explain what the above sentence means.

We can define a macro to return the maximum of two numbers as follows:

```
#define      max(a,b)      ( (a) + (b) + fabs( (a)-(b) ) )
```

where `fabs()` returns the absolute value of the argument. We then write a subroutine to calculate the minimum of two numbers:

```
double min( double a, double b )
{
    return( a + b - fabs( a-b ) );
}
```

A C program which, before being pre-processed, looks like

```
int main()
{
    double      x, y;
    double      maxValue, minValue;

    maxValue = max( x, y ); /* determine max. value */
    minValue = min( x, y ); /* determine min. value */
}
```

looks like, after pre-processing,

```
int main()
{
    double      x, y;
    double      maxValue, minValue;

    maxValue = ( (x) + (y) + fabs( (x)-(y) ) );
    minValue = min( x, y );
}
```

Except for argument substitution the pre-processor substituted in the definition of `max()` verbatim, but left `min()` alone since it is a subroutine. Also notice that the comments were stripped out by the pre-processor. The arguments to `max()` in the program are `x` and `y`—in the definition of `max()` we set up two *dummy* arguments, namely `a` and `b`. When the pre-processor substitutes in the actual code for the macro it takes the first argument in the actual call and substitutes this in wherever the first dummy argument occurs, and does the

same thing for all arguments. The actual call looks like `max(x,y)`, and we see that where the definition had `(a) + (b) + fabs((a)-(b))` the substitution into the program looks like `(x) + (y) + fabs((x)-(y))`.

If we made a mistake in formulating the definition of `max()` it is a simple matter to correct the entire program since we only need to change the macro definition and recompile. This is also true if `max()` were defined as a subroutine. Typically a macro is used instead of a subroutine if the code for the macro is short or speed is important.

Note that the definition for the macro (except for arguments) is substituted **exactly**. All text is substituted. C language statements are terminated by a semi-colon (;), but pre-processor statements are not. If `max()` had a terminating semi-colon then it also would have been included in the substitution, which would then look like

```
...
    maxValue = ( x + y + fabs( x-y ) );;
...
}
```

which is incorrect syntax.

A.2 Defining Constants

Another pre-processor directive looks very similar to a macro. This directive defines constants. As an example,

```
#define    NUMBER_OF_RCS_JETS    38
#define    NUMBER_OF_SS_ENGINES  3
```

Similar to macros, wherever the symbols `NUMBER_OF_RCS_JETS` or `NUMBER_OF_SS_ENGINES` are encountered in the program the values 38 or 3 are substituted verbatim. Notice the lack of a semi-colon.

A.3 Removing Definitions

If we define a macro or constant with a `#define` statement, we can later direct the pre-processor to disregard the definition. All subsequent references will then be undefined. The statement what we use is `#undef`. In the example above, we can un-define the `NUMBER_OF_RCS_JETS` symbol with the statement

```
#undef      NUMBER_OF_RCS_JETS
```

A.4 Including Files

When we write larger applications that must share definitions and declarations across many source files, we set up a separate file with just these definitions and declarations and include it in all our source code files automatically each time we compile. This is what has been done for the C library files. There are many include files defined for the C library. We direct the pre-processor to include them in our programs by using the `#include` directive. There are two slightly different forms of `#include`. The first surrounds the filename to include in angle brackets (`<>`) and the second form surrounds the name in double quotes (`""`),

```
#include    <stdio.h>
#include    <time.h>
#include    "myIncludeFile.h"
```

System library include files use the `<>` form and our application include files use the `""` form. The difference between the two forms is in where the pre-processor looks for the files. Files using `<>` are typically assumed to be in the system include library directories and files using `""` are assumed to be in the same directory as the source code. The C compiler will usually provide a method of specifying additional directories to search for files to be included.

When a file is included the `#include` statement is replaced by the contents of the file being included. Note that `#include` directives may be nested, with an include file having `#include` statements.

Include files, by convention, have a file extension `“.h.”` The `“h”` means *header* file. Notice in the example above that all three end this way. Using this convention allows for easy identification of include files.

A.5 Conditional Compilation Directives

The `#if`, `#ifdef` and `#ifndef` directives are used to conditionally keep or remove a section of code. If the `“if”` test returns true then the code is processed as normal C code, but if the test returns false then the statements contained in the directive are treated as comments and removed. The code contained in the test are all lines between the `“if”` and the ending directive `#endif`.

Often the `#if` directives are used to either include or exclude code that is computer-specific. For example, we can set up a `#if` directive block to set up some definitions depending on whether the system is a UNIX, MSDOS or VMS system. The code look like,

```
#define    UNIX        0
#define    MSDOS       0
#define    VMS         1

#if      (UNIX)
#define   BLOCK_SIZE   1024
#elif   (MSDOS)
#define   BLOCK_SIZE   512
#elif   (VMS)
#define   BLOCK_SIZE   2048
#endif
```

Another use for conditional code is embedding debugging statements in the software being developed. Surround the statements with

```
#ifdef  DEBUG
...
#endif
```

or better yet, specify debugging levels, with higher levels indicating more extensive debug processing,

```
#if    (DEBUG > 3)
...
#endif
```

Conditional code can also be used to write generic processors and include implementation-specific or department-specific code.

The three conditional directives are `#if`, `#ifdef` and `#ifndef`. The only directive with `#elif` capability is `#if`. The `#ifdef` and `#ifndef` test whether or not a symbol has been defined, while the `#if` test is a logical true/false. A source of examples for use of these directives is the include files supplied with the C language compiler and libraries.

A.6 Additional Directives

The last three directives are used infrequently, but are supported to provide flexibility and enhance implementation-specific information reporting by the language compiler.

6.1 Line

This directive is supported for various C program generators. The first argument is a constant positive integer value and instructs the compiler to take this number as the current line number for error diagnostics reporting. An optional second argument is a filename string which is taken to be the name of the current file being compiled, also for compiler error diagnostics reporting. The syntax look like,

```
#line integerLineNumber "filename"
```

6.2 Error

The error directive causes the compiler to display an error message that includes the information on the `#error` line. As with the `#line` directive, `#error` is seldom used in applications.

6.3 Pragma

The `#pragma` directive allows computer-specific directives to be coded. If the program is compiled on another computer which doesn't support the directive, it is ignored and no error message is produced. Consult the documentation for the compiler being used to find out what `#pragma` directives are defined, if any.

Appendix B

ASCII CHARACTER SET

Oct	Hex	Chr	Oct	Hex	Chr	Oct	Hex	Chr	Oct	Hex	Chr
000	0x00	NUL	040	0x20	Space	0100	0x40	@	0140	0x60	'
001	0x01	SOH	041	0x21	!	0101	0x41	A	0141	0x61	a
002	0x02	STX	042	0x22	"	0102	0x42	B	0142	0x62	b
003	0x03	ETX	043	0x23	#	0103	0x43	C	0143	0x63	c
004	0x04	EOT	044	0x24	\$	0104	0x44	D	0144	0x64	d
005	0x05	ENQ	045	0x25	%	0105	0x45	E	0145	0x65	e
006	0x06	ACK	046	0x26	&	0106	0x46	F	0146	0x66	f
007	0x07	BEL	047	0x27	'	0107	0x47	G	0147	0x67	g
010	0x08	BS	050	0x28	(0110	0x48	H	0150	0x68	h
011	0x09	HT	051	0x29)	0111	0x49	I	0151	0x69	i
012	0x0a	LF	052	0x2a	*	0112	0x4a	J	0152	0x6a	j
013	0x0b	VT	053	0x2b	+	0113	0x4b	K	0153	0x6b	k
014	0x0c	FF	054	0x2c	,	0114	0x4c	L	0154	0x6c	l
015	0x0d	CR	055	0x2d	-	0115	0x4d	M	0155	0x6d	m
016	0x0e	SO	056	0x2e	.	0116	0x4e	N	0156	0x6e	n
017	0x0f	SI	057	0x2f	/	0117	0x4f	O	0157	0x6f	o
020	0x10	DLE	060	0x30	0	0120	0x50	P	0160	0x70	p
021	0x11	DC1	061	0x31	1	0121	0x51	Q	0161	0x71	q
022	0x12	DC2	062	0x32	2	0122	0x52	R	0162	0x72	r
023	0x13	DC3	063	0x33	3	0123	0x53	S	0163	0x73	s
024	0x14	DC4	064	0x34	4	0124	0x54	T	0164	0x74	t
025	0x15	NAK	065	0x35	5	0125	0x55	U	0165	0x75	u
026	0x16	SYN	066	0x36	6	0126	0x56	V	0166	0x76	v
027	0x17	ETB	067	0x37	7	0127	0x57	W	0167	0x77	w
030	0x18	CAN	070	0x38	8	0130	0x58	X	0170	0x78	x
031	0x19	EM	071	0x39	9	0131	0x59	Y	0171	0x79	y
032	0x1a	SUB	072	0x3a	:	0132	0x5a	Z	0172	0x7a	z
033	0x1b	ESC	073	0x3b	;	0133	0x5b	[0173	0x7b	{
034	0x1c	FS	074	0x3c	<	0134	0x5c	\	0174	0x7c	
035	0x1d	GS	075	0x3d	=	0135	0x5d]	0175	0x7d	}
036	0x1e	RS	076	0x3e	>	0136	0x5e	^	0176	0x7e	~
037	0x1f	US	077	0x3f	?	0137	0x5f	-	0177	0x7f	DEL

Appendix C

C OPERATOR TABLE

Operator	Description	Type	Association	Prec.
()	Function call	unary	Left-to-Right	15
[]	Array element	unary	Left-to-Right	15
.	Struct/Union member	binary	Left-to-Right	15
->	Struct/Union Mbr. using ptr.	binary	Left-to-Right	15
!	Boolean not	unary	Right-to-Left	14
~	One's complement	unary	Right-to-Left	14
-	Negation	unary	Right-to-Left	14
++, --	Increment, decrement	unary	Right-to-Left	14
&	Address of	unary	Right-to-Left	14
*	Indirect addressing	unary	Right-to-Left	14
(type)	Typecast	unary	Right-to-Left	14
sizeof	Size of operand in bytes	unary	Right-to-Left	14
*, /, %	Multiply, divide, Remainder	binary	Left-to-Right	13
+, -	Add, subtract	binary	Left-to-Right	12
<<, >>	Left shift, right bit shift	binary	Left-to-Right	11
<, >	Less than, greater than	binary	Left-to-Right	10
<=, >=	Less/greater than or equal	binary	Left-to-Right	10
==, !=	Equal to, Not equal to	binary	Left-to-Right	9
&	Bitwise and	binary	Left-to-Right	8
	Bitwise or	binary	Left-to-Right	7
^	Bitwise exclusive or	binary	Left-to-Right	6
&&	Logical and	binary	Left-to-Right	5
	Logical or	binary	Left-to-Right	4
?:	Conditional	binary	Right-to-Left	3
=	Assignment	binary	Right-to-Left	2
* =, / =, %=	Shorthand operate/assign	binary	Right-to-Left	2
+ =, - =				2
<<=, >>=				2
&=, ^=, =				2
,	Multiple expressions sep.	binary	Left-to-Right	1

BIBLIOGRAPHY

- [1] AT&T Bell Laboratories. *The C Programmer's Handbook*, 1985. Published by Prentice Hall.
- [2] Boillot, M. *Understanding Fortran 77*. West Publishing Co., 1984.
- [3] Clark, R. and Koehler, S. *The UCSD Pascal Handbook*. Prentice Hall, 1982.
- [4] Coad, P. and Nicola, J. *Object-Oriented Programming*. Yourdon Press, PTR Prentice Hall, 1993.
- [5] Dewhurst, S. and Stark, K. *Programming in C++*. Prentice Hall, Second edition, 1989.
- [6] Kernighan, B. and Ritchie, D. *The C Programming Language*. Prentice Hall, Second edition, 1988.
- [7] NeXT Computer, Inc. and Merriam-Webster, Inc. *Webster's Ninth New Collegiate Dictionary and Webster's Collegiate Thesaurus*, 1992. First Digital Edition, Version 3.0.
- [8] Prentice Hall. *Unix System V Release 4*, 1990. Programmer's Guide: ANSI C and Programming Support Tools.
- [9] Sowell, E. *Programming in Assembly Language: Macro-11*. Addison Wesley, 1984.
- [10] Stroustrup, B. *The C++ Programming Language*. Addison Wesley, 1991.