C Language User's Handbook

by
Weber Systems Inc. Staff

Ballantine Books • New York

Contents

Preface

This text can be categorized as a primer or tutorial. Its express purpose is to introduce its owner to the C programming language. It was written with the following fundamental assumption:

- The user is familiar with the very rudimentary concepts of computer hardware and software.

In addition to the aforementioned prerequisite, it is recommended that the reader have access to a computer and compatible C software. While there are numerous program lines and several complete programs presented in this text, there is no better way to learn a language than through practice.

While this manual was designed to teach even the computer neophyte basic C programming, the author does not endorse learning C as a beginning language for two reasons. First, C's unusual structure, power, flexibility, and affiliated standard library may overwhelm the novice. Secondly, C is a

17

relatively free and forgiving language. As such, it does not force one to develop structured programming techniques and concise syntax usage. Without proper discipline, it is extremely easy in C to end up with undecipherable code instead of an orderly, modular program. Also, all things held equal, readers familiar with another high level language will have an easier time learning C.

Considerable effort has been taken to present the topics covered in this text in a clear and concise manner. With the possible exception of chapter 2, "An Overview of C", all chapters are presented in order of learning precedence. Unfortunately, there are instances where concepts are presented prematurely in order to demostrate essential aspects of the C language. These occurrences, while unavoidable, have hopefully been kept to a minimum. Wherever they appear, a reference to the supporting chapter is present.

Additionally, this book contains sections whose headers are appendaged with the word advanced enclosed in brackets (i.e. [Advanced]). It is suggested that readers with a minimal knowledge of C either completely ignore or skim these sections on the first reading. Special attention should be paid to sentences where either individual words or the entire sentence is *italicized*. Such sentences contain key information, usually on the concepts underlying C. When individual words are italicized, that concept is generally defined at that point in the text. **Boldface** type is reserved for words that form either the C language or are used throughout this text in example programs.

Although, as previously mentioned, this text is instruction oriented, it may be used as a reference manual, albeit a wordy one. It contains the majority of traits inherent in conventional C language. The definitive text on C, *The C Programming Language,* cowritten by the creator of C, Dennis Ritchie, makes a more succinct reference guide but is considered by some to be a very terse tutorial.

The rest of the book is organized into 11 chapters, 6 appendices, and an index. Chapter 1, Introduction to C, deals with the history and relative traits of the Unix operating system and the C language. It also presents the reference system and discusses program usage.

Chapter 2, "An Overview of C", uses an example program to illustrate C convention, construction, and syntax.

Chapters 3 thru 5 present four fundamental aspects found in any computer language: primary data types, operators, control flow, and elements of program design. It is essential that readers familiarize themselves with this material before continuing to later chapters.

Chapter 6 will expand on the concept of functions in C. This chapter will explain the process of call by value, a pivotal concept in the mastering of the C language. It is by this process that functions normally communicate with each other.

Chapter 7 will delve into storage classes. These divisions of data are sanctioned under the use of multiple functions. The scope rules inherent in these divisions are detailed, and the proper use of each class is discussed.

Chapter 8 touches on some general traits of C compilers, and also introduces the C #define and #include utilities.

Chapter 9 deals with pointers and arrays, while chapter 10 discusses structures, unions, and bitfields. All of these are derived from the primary data types of chapter 3, and as such are handled distinctively.

Chapter 11 advances input/output (I/O) operations and summarizes other typical standard C library functions.

Text Conventions

In this book, most long programs will be printed in dot matrix type. Program output will be in similar type. However these sections will be enclosed in rectangles to differentiate them. When a portion of a program appears in italics as shown below:

```
int total = integer value;
```

The italicized portion does not indicate actual code. Rather it indicates a general representation of a portion of the program. This symbolic portion represented in italics could be replaced by any one of a number of actual sections of code.

Program code referenced in text will be printed in **bold-face**. Boldface will also initially be used to indicate C keywords and functions, but thereafter keywords will be printed in a type style called megaron medium.

Within the text, entire functions will be symbolized by the function identifier immediately followed by a matching pair of parentheses. The formal arguments will not be included within the parentheses. For instance, main () will be substituted for "the main () function".

1

Introduction to C

Origin of C

C was developed in the early 70's by Dennis Ritchie, a systems software engineer at Bell Laboratories in Murray Hill, New Jersey. Although C was designed as a language to be used with the Unix operating system, it soon proved itself to be a powerful, general purpose language. In a very real sense Unix shaped C to its own purposes. In order to place C in its proper perspective, our discussion will begin with an overview of computer languages, continue with a brief history of Unix, and then conclude with a discussion of the development of C.

Overview of Computer Languages

To fully appreciate the impact of C on the Unix operating system, one must know something of the hierarchy of computing languages. A chronological review of these languages is given in figure 1.1. We will review these languages in the remainder of this section.

MOST RECENT

Artificial Intelligence Dialogues
(Reasoned Learning)
↑
Command Languages
↑
Low Level or Machine-oriented Languages
↑
High Level or Problem-oriented Languages
↑
Assembly Languages
↑
Machine Languages
↑
Mechanical Hardware
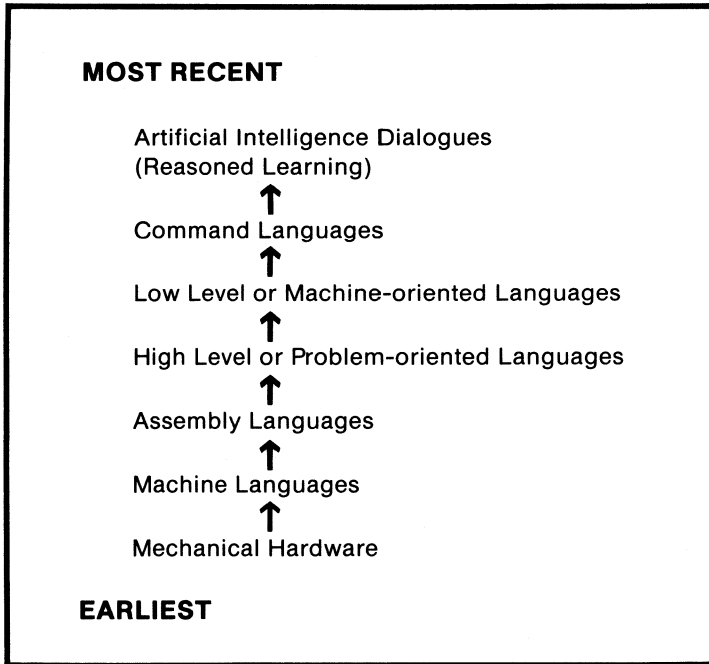
EARLIEST

Figure 1.1 Chronological review of computing languages

The earliest category in figure 1.1, mechanical hardware, includes early predecessors of the computer such as Charles Babbage's Analytical Engine (1834) as well as recent mechanical adding machines (typified by powered or manual crank-type cash registers). These machines are quite inflexible and their language consists of physical movements and processes.

Modern computers operate through analytic logic based on the two binary states, true and false or 1 and 0. The microprocessor or central processing unit (CPU)*, where all actual operations are performed, has its own language consisting entirely of fundamental instructions represented by binary strings. For example, the add instruction for the 8088 microprocessor family is 10000011 B (B for binary). Programmers of the first electronic computers were required to program in machine code. As you can probably imagine, this significantly decreased the productivity of said programmers. In order to increase programming efficiency, more user-friendly languages had to be developed. However, it is important to remember that the CPU can only directly handle machine language, and all higher languages must be translated into machine code.

Assembly language is the first stage of human compatible or higher level languages. The logical successor to machine language, assembly language has exactly the same formation and true capabilities but includes English commands and labelling, remarks, program format, and alternate base 8, 10, and/or 16 usage. For instance, the "add" command in assembly language is immensely preferable to the analogous machine language command. Even so for many applications, coding in assembly language is excessively tedious.

---

*In microcomputers the CPU and the microprocessor are synonymous entities. This is often not true for larger computers.

The next family of languages to be developed were the high-level or problem-oriented languages. The most familiar of these include BASIC, Pascal, FORTRAN, LISP, and others. These languages were developed in order to handle external problems (problems not involving system function) in fields such as business, mathematics, science, engineering, etc. They are rich in control flow structures and special operations. They have few "down to the machine" features such as pointers, defines, increment operators, bitfields and bitwise operators (all of these concepts are introduced later in this book). Recognizing this void, computer researchers decended a rung and focussed their talents.

Low level or machine-oriented languages were the result of this redirection of effort. These languages are used in systems environments — to write data processors, compilers, operating systems, etc. They were used to handle processes and problems associated directly with the operation of the computer as an integrated system. C's ancestry may be traced directly to this class of languages.

Command languages deal with the use of the predefined functions of the *command shell* of the operating system. Logging in; entering edit, language, or other mode; input/output device and directory usage are all illustrations of command language utilization. The *shellscripts* of Unix are exceptional examples of how powerful such an arrangement can be. Command dialogues (here a dialogue refers to an interactive language implementation) will assume increasing importance in the future as they are tailored to flexible industrial robots.

At the top of figure 1.1 is the broad category labelled "artificial intelligence dialogues". Artificial intelligence refers to the *simulation* of the capacity for *human* thought process (reasoning and learning) in a construct (here a computer). The term "artificial" may be somewhat of a misnomer when applied to these "smart" machines for two reasons. First, it may very well be reasonable to expect that one day computers may be

endowed with "true intelligence" although intelligence is an elusive concept. Secondly, such a machine need not mimic the thought processes of humankind. It is only due to our ignorance of other possible modes of thought that researchers mainly rely on such ethnocentric models.

Commerical products from this field will make it possible for users to communicate in a more free and natural way with a computer system. In some applications, speech recognition and synthesizing systems will allow the most effort-free interface communications (short of telepathy) possible. The talking machines of science fiction will someday be fact.

History of Unix

The story of Unix and later C is a tale, like so many others in areas of rapidly developing technology, of improvisation and innovation in the face of need. In the late 1960's, Bell Laboratories was partnered with the Massachusetts Institute of Technology (MIT) and General Electric in a project to develop the first *interactive, multi-user/multi-tasking* operating system. This system was to be implemented on General Electric's 645 mainframe computer (GE is no longer a computer manufacturer). *Multi-user* refers to a system that allows several customers the use of a computer while the *multi-tasking* capability enables such use at what appears to be the same time. *Interactive* refers to a system that supports immediate dialogue with a user (as opposed to batch I/O operations involving devices such as card readers, magnetic tape, etc.). Although the resulting system, *Multics,* was a techical success, in that it accomplished the project goals, still that initial incarnation of Multics proved too slow and cumbersome to be an effective tool. In 1969, Bell Labs decided to pull out of the project.

Bereft of the capabilities of Multics, a young computer scientist, Ken Thompson, working in the computing science research department (Murray Hill, NJ) of Bell Labs was forced to move his work to a more affordable minicomputer, the Digital Equipment Corporation (DEC) PDP-7. Since at this time there was a dearth of software for the PDP-7, Thompson created not only an assembler but a new file system and primitive operating system to control the system. In a pun on Multics, the system was dubbed Unix by its lone inventor, Thompson.

Thompson fashioned Unix to be a software programmer's workshop. In addition to the *kernel* or core, whose job it is to handle normal operating tasks and data storage, the system also has incorporated a large *utility library* and a voluminous *command shell* that allows a high degree of control over the system. In fact one can program in just these modular commands, creating *shellscripts*. Unix's other outstanding features include multiuser/multitasking capabilities, a hierarchical file system, I/O independence (input/output device utilization is not "locked" into the operating system), I/O redirection and *piping* capability. Although numerous enhancements to Unix were implemented by both Ken Thompson and newcomer Dennis Ritchie, perhaps none was more important than recoding Unix in Ritchie's brainchild, C.

C's Ancestry

Unix was orginally written in assembly language for the PDP-7. When the more powerful PDP-11 was procured for their research, Thompson and Ritchie recoded Unix for that mini-computer. This version of Unix found commercial use in other departments of Bell Labs. The process of adapting Unix for a new computer illuminated a very considerable obstacle in

software, namely that of *portability*. Portability refers to the ease in which software from one computer system can be moved to others. Assembly languages are very nonportable entities because they are wedded very closely to the architecture of the microprocessor. Since there are many different microprocessors in use, and since new ones are constantly being introduced (with little industry standardization), assembly languages tend to be nonportable and relatively short-lived.

At this time, Thompson was also working on improving programming languages relating to their use in operating systems. For a time, he worked with Basic Combined Programming Language (BCP), a machine-oriented language designed by Martin Richards at Cambridge in 1967. BCPL is itself a simplication of the verbose Combined Programming Language (CPL). CPL, also a machine-oriented language, was developed in the early 1960's at the University of London and at Cambridge. CPL, in turn, borrows many of its features from the problem-oriented language ALGOL 60. Thompson consolidated BCPL down to its basic features and dubbed the resultant interpreted language "B". B, however, had a number of grave shortcomings. Though adequate for the hardware available at the time, B was essentially a very small subset of CPL and as such, had limited applications. Also B, like its two immediate predecessors, was a machine-oriented language. Machine-oriented languages have historically had poor reception from applications programmers because generating code in such languages is more difficult.

Dennis Ritchie salvaged the best features of B, changed some of its more restrictive details, added features to it (such as data types and storage classes), and lifted the language above hardware specifics. C was born. C shares some important attributes with Unix. Since both were created by individuals, they have a coherence and continuity that committee inventions often lack. They are in essence very simple, yet their basic

components can be worked in a modular fashion to build units of great authority. Both are flexible and relatively fast.

Advantages and Disadvantages of C

Some of the relative advantages of C have been briefly mentioned. C is often described as a "robust" language. C has a wide diversity of operators and commands that make it a very general-purpose language. Dennis Ritchie, however, designed C to be used in system software environments - operating systems, compilers, text processors, data base management, etc. C is probably best suited for these functions. Paradoxically, a rapidly growing applications program market for C has also developed.

The key to this seeming enigma lies largely in C's rather unique position in the hierarchy of languages. C occupies a position between the low-level or machine-oriented languages and the high level or problem-oriented languages. C is sometimes referred to as a middle-level language. C's low level attributes allow one to specify logic detail, manipulate heap memory, effect bit and other low level operations. C's high-level characteristics are evidenced by its advanced control structures, function usage, data types, etc. C is close enough to the hardware to allow extensive control over program implementation. In this way one can realize significant increases in computer efficiency. Yet C is removed enough to hide the specifics of the computer architecture.

C is also clear, compact, pragmatic, and concise. C shares these attributes with other single inventor languages such as APL and Pascal. How can C be robust, as mentioned before, and compact concurrently? The answer lies in the way Ritchie, using B as a model, relied on a lean set of basic tools which could be combined using simple but permissive rules to build complex structures.

C does not have special functions (i.e. trigonometric and geometric functions) built into the language. In this sense, it is lean. It is nonetheless robust in that it includes many basic operators, some of which many common languages lack. For example while C does not inherently possess the square function (x) as does BASIC, such a operation can be performed with an algorithm using simple math operators (This indeed is how it is accomplished in BASIC, the process is just invisible to the user). On the other hand C includes operators not available in BASIC. Like children's building blocks or the sequence units of the DNA molecule, C's potential lies in its use of modular components to create more specific and powerful constructs (programs and functions).

C, because it is new and has weathered its formative years inside one consistent environment (Bell Labs), has avoided the permutations and alterations that other languages, such as BASIC and Pascal, have undergone. One can talk of a standard C as defined by Kernighan and Ritchie's text *The C Programming Language.*

C's uniformity and middle level hierarchial position combine to form another favorable attribute — *portability.* Remember portability refers to the ease in which a unit of software or hardware may be moved to another computer. Often a program may be run on another system with very little modification. This is impossible to do with assembly language programs and more difficult with some machine-oriented language programs, because these languages are tied more closely to specific machine architecture.

Industry reliance on CPU-specific assembly languages for product development can be a short term proposition. In the days when hardware was relatively much more expensive than a programmer's time, this situation was acceptable. As the price of hardware inevitably falls, the process of modifying software for each successive model of microprocessors becomes

prohibitive in terms of retraining costs, programming time, product lag time, and reliability.

The C language is rapidly gaining popularity. One reason for this growth is the fact that Unix is written in C. While this may not directly concern a great number of readers, it will have a large general market impact. It is estimated that by the year 1987 there will be 1.4 million licensed Unix sites.* Unix and Unix look-alikes will be implemented in the commercial environment. This will necessarily create a substantial demand for programmers not only familiar with Unix but fluent in C. While C's acceptance was initially linked with Unix, as compilers compatible with other operating systems appear, C is gaining favor through its own merit.C is being used in other software applications such as writing other operating systems and compilers (C compilers are largely written in C). Likewise this will create markets for C programmers.

Having catalogued many of C's positive features, let's turn to the other side of the coin. What are some of C's negative points? Since more than a few traits are two-edged swords, let us begin by reviewing, in a different light, some of the advantages of C.

Earlier, we stated that C was a flexible general-purpose language. Although the Swiss Army knife is a excellent all-purpose tool for the campsite, it is not generally used for carving the Thanksgiving turkey. In the same vein, if one only wants to perform a single set of very specific tasks, then C is probably not the most well-suited language.

C has a very high execution speed. But, as previously mentioned, to obtain executable code one must first compile the program in question. Depending on the compiler and the amount of necessary debugging, this can be a tedious and time consuming job. For short application programs, it is often

*Unix and the standardization of small computer systems,by Jean L. Yates. BYTE Oct. 1983

much quicker from a program design standpoint to make use of an interpreted language. The additional execution time is usually not critical.

Another disadvantage of C is its tendency to consume considerable amounts of memory. It is not unusual for compiler implementations to use 64K of main memory. C executable file lengths can also be of inordinate lengths. The reasons for this will be discussed later in this book. A program that just prints a line can occupy over 12K of memory for a fully supported C implementation. Some of this usage can arise from the inefficiencies of implementing C on non-Unix operating systems. C was not designed with memory efficiency uppermost in mind, and this problem is magnified when C is used for short applications programs. With such programs, it is probably wiser to use an interpreted language.

C only has single-thread control flow. Operations such as multiprogramming, parallel operations, and co-routines are not supported in this language (These operations allow "simultaneous" execution of two or more sections of interdependent code). To have built in such capabilities would have greatly complicated the language and compiler. Such capabilities would extend C's utility only into a very restricted and advanced area of programming.

A number of other minor criticisms of C have been put forth. These include:

- C sacrifices power for freedom
- C includes no operators to deal with composite objects such as strings and arrays
- C does not provide for heap garbage collection

Reference Hardware and Software

All programs in this book were run on the IBM XT® using 256K of RAM, a standard double density 5¼ inch floppy and a 10MB hard disk drive. (Alternately, the dual floppy drive configuration may be substituted.) Some compilers can be run on the one drive configuration. However, we recommend using the two drive configuration. For larger programs additional memory may be needed.

The standard operating system, MS(PC)™ DOS 2.0, was used throughout. Although C generally operates best on Unix and Unix workalikes, it is by no means necessary to have such an operating system. Many compilers exist for implementation on non-Unix systems.

All programs included in this text were run on Computer Innovations Inc. C86 C Compiler*, and/or on Lifeboat Associates' Lattice 8086/8088 C Compiler.** Both compilers support full Kernighan and Ritchie C (except both compilers convert **register** to **auto** data type, q.v. chapter 7), and both possess large subsets of the standard C library. Furthermore both supplant a fair number of useful standard Unix system calls with nonstandard C library subroutines. The compilers require 96KB and 81KB of RAM respectively. This includes an allowance for the DOS operating system. A special thanks also to the people at C-systems*** for the use of their fine compiler which includes the very useful C-window Debugger.

---

*Computer Innovations, Inc., 75 Pine Street, Lincroft, NJ 07738 (version V1.33D).

**Lifeboat Associates, 1651 Third Avenue, New York, NY 10028 (version 1.04)

***C-Systems, Box 3253, Fullerton, CA 92634

While C was created to serve as a systems language, it is always easier to introduce a new language from an applications point of view. The programs are naturally more independent, self-contained, and to the vast majority of people, easier to follow. The programs presented are not typically written efficiently from a programming point of view. They are also, in a large way, contrived or artificial in that the programs have very little utility (for example most do not accept any input). Their purpose is to demonstrate important aspects of the C language. They are not meant to be representative of common code.

As noted in the preface, certain sections of this book are labelled [Advanced]. The C novice can consider these sections nonvital for learning basic C. Still it is suggested that such a reader at least familiarize himself/herself with the existence of these concepts. For example, in chapter 2, the section entitled Function Calls briefly touches on information fully presented in chapter 6. This section was included in order to give the reader a broader feel for C and, perhaps more importantly, to acquaint him with an essential concept that will be covered later.

2

An Overview of C

Introduction

This chapter is intended as an introduction to the components, syntax, and style of C. Most of the subjects alluded to will be covered in greater depth later in this book. Some of the conventions generally used in C and the conventions specific to this text are detailed in this chapter. Many of the concepts covered in this chapter are illustrated in the example source program, aptly named Program 2.1, presented in figure 2.1.

```
            /* Program 2.1 */
/* This program prints several characters and a char-
    acter string to the standard output device */

main()                   /* line #5 */

{ char a, b, d, c;
    a = 'T';
    d = 'h';
    b = 'e';             /* line #10 */
    c = 'n';
 printf("\n Program 2.1 prints-");
 printf("\t%c%c%c %c%c",a,d,b,b,c);
 a = 'd';
 printf("%c",a);     /* line #15 */
}
```

```
Program 2.1 prints-    The end
```

Figure 2.1. Program 2.1 source listing and output

Language Composition

All high-level languages are composed of simple units. For example, spoken English consists of phrases which can be subdivided into words and still further into syllables and their component consonants and vowels. There are extensive, and in the short run, inflexible rules which must be followed in order to form correct phrases. (Speech is, of course, much more complicated than this. For example, rhythm, intonation, and inflection can alter the meaning of a phrase.) Contemporary computer languages require a less complicated but more definite structure, as they must be translated into a specific binary code sequence for the CPU.

The C language is composed of seven basic types of code: *constants, variables, operators, punctuation, keywords, function identifiers,* and *labels* (collectively known as *tokens*). Specific syntax rules allow these components to be joined in a logical fashion to accomplish specific jobs. Although C is a relatively free language, one actually has little control over either the form these basic components take or over the ways in which they may be combined. *Comments* or *remarks* are also permitted, however they are not properly a part of the language.

CONSTANTS

Constants can be divided into two categories: *non-numeric* and *numeric literals (constants).* A non-numeric literal constant may be composed of letters, numerals (jointly referred to as *alphanumeric characters*), blanks, and any special printing or nonprinting characters supported by your system. Since a computer can only manipulate *binary* data, all characters must be encrypted or encoded into a binary code. Binary refers to the base two numbering system. Appendix A contains a binary/octal/decimal/hexadecimal/BCD conversion table. The majority of all computers use the encoding

system called the *American Standard for Coded Information Interchange* or *ASCII* (pronounced "as'key"). Appendix B lists ASCII character codes.

Literal constants in C, depending on the application, are enclosed in either double or single quotes. In figure 2.1, lines 8 through 12 and line 14 all contain literal constants. Program 2.1 contains five single-value literals enclosed in single quotes. These are referred to as *character values*. In line 12, the following literal:

Program 2.1 prints-

is enclosed in double quotes. This literal is an example of a character string. Character strings and character values are handled quite differently in C.

Numeric data in C is primarily made up of numbers but can include decimal points and certain letters. Numeric constants can be differentiated from non-numeric constants by the absence of quotation marks. All such input is automatically converted to base two for storage or processing. In C, a numeric constant can assume one of several forms or classes. These numeric classes in addition to the character class comprise C's fundamental data types (chapter 3).

Valid Constant	Invalid Constant	Reason for Invalidity
1957	13J	non-numeric
';'	'A;'	more than one character value
"hello"	""hello""	ambiguous quote usage*
'\"'	'''	ambiguous quote usage*
13.473	19+3i	invalid numeric class
.40	2/5	designates mathematical operation not constant

* These situations are discussed in chapter 3.

VARIABLES

Variables can be conceived as empty boxes that can accept or store any one of a number of possible values. The contents of a variable are normally changed via operators within the program. In C, each fundamental data type has a corresponding variable type. C also allows construction of many derived data types, the most rudimentary of which will be introduced in the second part of this text. Because the compiler and the CPU handle these data types in distinctive manners, *it is necessary to declare variables before they are used in a program.* When a variable is declared, it is designated as belonging to one of the data types. In Program 2.1, variables **a, b, c**, and **d** are declared to be of character (**char**) type on line 7.

Initialization is the process of *assigning* a variable its first value. As a rule of thumb, all variables must be explicitly initialized in the program to be useful. In our variable-as-a-box model, an analogous statement would be that a box derives its purpose through what it carries and consequently empty boxes are of little use. Variables are assigned values either directly as in:

x = 23;

or through the use of one or more operations:

x = (y * 4)/100;

For this second assignment expression to be valid, **y** must have been previously initialized. Lines 8 through 11 in Program 2.1 are examples of initializtion. Here, the variables are assigned character values. In line 14, variable **a** undergoes reassignment; its original value, **T**, is replaced by the value, **d**.

A variable may be assigned a value which never changes throughout the program. This can be a convenient tool. It is far easier to write **PIE** once it has been assigned an appropriate

value, like 3.14159, as opposed to continually entering the above numeric literal value. In Program 2.1, all variables except **a** retain constant character values. Although such variables are often loosely referred to as "constants", their proper appellation is *symbolic constants*. C allows for a more convenient means of referencing symbolic constants. This method will be discussed in chapter 8.

OPERATORS

Operators inform the computer what tasks it is to perform as well as the order in which to perform them. C's operators are easily distinguished because they consist of one or more special printing symbols. Here is a complete roster.

( )	[ ]	->	.	!
~	+ +	- -	-	*
&	sizeof	(*cast*)	*	/
%	-	<<	>>	<
<=	>	>=	= =	!=
|	&&	||	? :	=
+ =	- =	* =	/ =	% =
>>=	<<=	& =	^=	| =
,	^	+		

Notice that, apparently, some of the operators have been listed twice (e.g. ( ), -, *, and &). These symbols can denote different operations depending on the context of their usage. Fortunately, with experience, the C programmer will generally have little difficulty segregating these identical operators.

In Program 2.1 the only operator used is the assignment operator, = . For example, in line 8, the value within the single quotes on the right side, **T,** is assigned to the variable **a** on the left. We may interpret the assignment operator by substituting

the phrase "is given the value" in its place. Operators will be discussed in detail in chapter 4.

LANGUAGE PUNCTUATION

Punctuation is used in C, as it is in written modern languages, to better define structure and to separate constituent parts. C's punctuation symbols are as follows:

$$\{\} \quad () \quad "" \quad '' \quad /* */ \quad ; \quad , \quad whitespace$$

Two of these, the parentheses and the comma, serve basically the same function they do as operators (i.e. to associate and to separate respectively). They are included here because they also lend structure to expressions in C. The various punctuation marks available in C are listed in table 2.1 along with their associated functions.

Table 2.1. C Punctuation Marks

Puntuation Mark	Description
braces { }	Delimits function sections or blocks
parentheses ( )	Groups related items, identifies expressions
double quotes " "	Specifies and delimits character strings
single quotes ' '	Specifies and delimits character values
backslash asterisk pairs /* */	Specifies and delimits comments
semicolon ;	Denotes end of simple statements (terminator)
comma ,	Separates related items
whitespace or blanks	Separates alphanumeric code (function identifiers variables, keywords, etc.), improves readability

COMMENTS

Comments, or remarks as they are known in some other languages, are supported under C. Any group of characters situated between a forward slash and asterisk matched pair is considered to be a comment. Comments are invisible to the C compiler; that is, they are not translated into object code. Most C compilers do not allow comments to be located within statements. Usage of comments in such a manner (as illustrated below) is inadvisable in any circumstance.

a = /* assigning T to variable a */'T';

Comments are a very useful documentation tool. Proper documentation is an essential programming habit. Appropriate documentation helps insure that a programmer will be able to quickly grasp his or her program's logic, even though he or she has not handled it in months. Effective documentation also enables a programmer to understand a program written by another. Minimum program documentation requirements would include the following:

- Descriptive introductory comment for each function.
- Program identification and synopsis prior to main( ).

In Program 2.1, lines 1 through 3 demonstrate the use of comments to identify the program (which is wholly comprised by the main() function) and summarize its purpose. Lines 5, 10, and 15 also contain line identifier comments.

A comment can extend over several program lines. In Program 2.1, a single set of delimiters could have been used to identify these lines as one comment. *Comments may not be nested* (i.e. placed one inside another) as in:

/* Program 2.1 /* written 1/84 */   */

KEYWORDS

Certain identifiers are reserved for use as keywords. A complete list of C keywords is given below. All keywords, except **entry**, and **sizeof** either serve in a *declarative* or *control flow* capacity. Sizeof is an operator while entry is a undefined keyword available for future implementation.

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

All keywords (except entry) enable or implement specific, unique operations or processes. A keyword must be specified precisely. Approximations are not allowed — neither **registers, character,** nor **goto1** are regarded as keywords in C. Therefore, *every keyword in C must be separated from prior and subsequent alphanumeric code with at least one blank (whitespace).*

LABELS

Labels are used to uniquely identify lines chosen as destinations of **goto** statements (q.v. chapter 5, Control Flow). Goto statements are used infrequently in *structured programming.*

Structured programming is both a school of thought and a programming style. It stresses a disciplined approach involving top-down design (starting with the most general and successively leading into more detailed steps), use of self-contained section structure, indentation, and extensive documentation via comments.

Functions in C

It has been repeatedly mentioned that one of the most useful aspects of C is its modular design. C programs are constructed [almost] entirely of these modular units, called functions. A function is a set of definite operations that are performed on required data string or numeric values. The most common functions are usually mathematical in nature, such as the square root or sine functions on a calculator. Functions are *defined* (i.e. composed and characterized) by one or more *statements* designed to accomplish a particular task. Functions are composed of two parts: a *function header* and a *function body*.

In Program 2.1, the line main() is the function header. The function header is itself composed of two parts: main and (). main is termed the *function identifier* or more simply, the function's name. The parentheses, as well as the characters they enclose, comprise the *formal argument*. (These enclosed characters are usually variable names.) This group of characters, along with any delimiting commas, is known as the *formal argument list*.

As described in chapter 6, formal argument lists are the vehicles through which information is received from other functions. main() has no formal arguments (or alternately it has a *null* argument). The main() function is passed no initial information from other functions. If one were to create a function that inputs one value and squares it, its header would look something like this: squ(x). The single variable x is the only formal argument; thus the function squ(x) may only be passed one input value per use. All of the programs prior to chapter 6 will be single *user-supplied* function programs with a null argument list.

The second section of a function, the function body, consists of a number of statements delimited by a matched pair of braces — { }. The function body is the meat of the function. It is

here that all of the actual work is performed. The length of this section is theoretically only limited by the maximum file length allowed by your system. However, in practical use, source files rarely approach this limit due to the simplicity of *linking* smaller subprogram files (chapter 8). Only program statements and comments may appear within the function body.

FUNCTION IDENTIFIERS

A function identifier, or name, can be easily recognized due to its syntax and its position outside the function body. A special type of function identifier can also be located within the function body. These are referred to as *function calls*. A function call is also followed by a matched pair of parentheses, and it can also include an argument list. *A function call temporarily transfers program control along with any required information to the function named in the call.*

In the example program, printf() is a function call. (Do not concern yourself at this time with the details of printf()). Since there are several expressions in C that require parentheses, a C programmer must learn to differentiate between them. A function call must begin with a valid name (chapter 8). A matched pair of parantheses, optionally enclosing a list of arguments must follow. A function name may not duplicate a keyword.

Statements

A statement is a section of code that, when translated, directs the computer to carry out a specific *instruction* or group of related instructions. (Comments are sometimes also considered statements. This discussion will ignore them.) The term instruction refers to any basic, complete process allowed in the reference language. The following are examples of instructions in C:

- Adding two numbers and assigning the result to a variable
- Calling a function routine
- Comparing two values and performing some single operation on the outcome of that comparison

Each statement may include a number of elementary expressions. An elementary expression, as the word is used here, is the most elementary language or system process possible at the reference level of programming. Therefore, adding two numbers is a elementary expression, another would be assigning that result to a variable. If these two related elementary expressions are coded in a continuous, connected manner, the result will be an instruction. By supplying proper punctuation, an instruction can be transformed into an acceptable source-level statement.

The following example should help clarify the concepts of statement, instruction, and elementary expresion. First, an addition and an assignment elementary expression will be used to construct an instruction.

b+3,   *and*   **a=( )**   ⇨   **a=(b + 3)**   *Optional Parentheses*

The addition of correct punctution transforms this instruction in a statement.

a=(b+3);

C allows its users to combine elementary expressions into fairly complex instructions.

a = b + 3 = ((c + 3) / (a * a) - 5

Complex instructions are allowed only if they are continuous and nonambiguous in nature. For example, the following instruction:

a = b + 6   d = a/b

would not be allowed as the compiler would attempt to "read" the line as one statement. This line does not qualify as one statement because it is composed of two separable, distinct instructions. Upon its inability to translate such a line, the compiler would generate an error message such as: "bad syntax" and/or "semi-colon expected." These two instructions could have been combined to form the statement:

$$d = (a = b + 6)/b;$$

Statements may be divided into four main categories: *declaration, function calls, operationals,* and *control flow (conditionals and loops).*

We have already discussed declarations (Program 2.1, line 7) and function calls (lines 12, 13, and 15). Lines 8 through 11 and line 15 are operational statements using the assignment operator =. A statement which includes an assignment operator is known as an *assignment statement.*

Conditional statements are hybrid statements composed of a *relational condition* expression in conjunction with a statement of one of the above types. A conditional statement, as the name implies, will perform the indicated instruction(s) only if a specified condition has been met.

Loop statements are likewise hybrid statements, however the contained instruction(s) will be executed a number of times indicated by the loop condition. Multiple execution of the same instruction or statement is called *iteration.* Conditional and loop statements are presented in chapter 5.

Semicolons

In Program 2.1, all statement lines end with a semicolon. Also, notice that the other lines, 1 through 6 and line 16, do not contain semicolons. A semicolon can only be used as follows in a C program:

- Following a statement as its delimiter
- Within a comment, where it will be ignored by the compiler
- Within the control flow expression known as the **for** conditional statement (see chapter 5)
- Within single or double quotes where it is used as a character value or element of a character string, respectively

For now, it is important to remember that *all simple statements must end with a semicolon.* Non-statements generally do not possess a semicolon.

For example, the placement of a semicolon at or beyond column 7 on line 5:

main( ); /* line #5 */

would instruct the compiler to treat that line not as the function identifier **main( )** but as a function call to **main( )**. To complicate matters, the function would then have no identifier, and the call to **main( )** would be invalid.

The main( ) Function

Since a C program can be fashioned from many different functions one might ask, "in what order are these functions to be executed?" When a complete program is compiled in C, the **main( )** function marks the beginning of execution. *Therefore, every complete program must have one and only one* main() *function.*

The main() function also normally directs the flow of action. Execution begins with main(), progresses through it sequentially, branches off to other functions as instructed, and normally ends with the last statement in main(). Because of the control exercised by the main() function, it is often referred to as the *driving function* or the *driver*.

Inasmuch as C requires that main() be the function executed initially, it generally does not include a formal argument list (i.e. other than a null list). The reasoning behind this is that since main() is, by definition, the first active function, then no other function could be active before main() in order to pass data to it. Data "waiting" to be read from another file or from a *pipe* cannot be passed to main() utilizing header arguments. Instead, C includes special library functions to accomplish this task. Some operating systems, Unix for one, have a number of commands that also deal with inter-file and program communication.
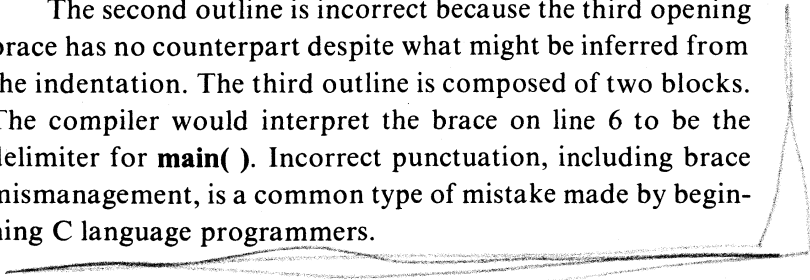
Braces

Braces are used to mark the beginning and ending points of the function body. A section of code enclosed in braces is said to be a *block*. By definition, a function body is then a block. A function may also be subdivided into blocks for any number of reasons.

When nesting braces in this way, the compiler matches the first brace with the last brace (function body delimiters), the second brace with the second to last brace, etc. A function can only be defined by one all-encompassing block which can be subdivided further.

Refer to figure 2.2. The braces in 2.2a are arranged correctly. Each opening brace { is matched with a closing brace } (here, matched pairs are directly underneath one another). Also, all code (the dots represent any number of statements) is enclosed by the first brace pair.

The second outline is incorrect because the third opening brace has no counterpart despite what might be inferred from the indentation. The third outline is composed of two blocks. The compiler would interpret the brace on line 6 to be the delimiter for **main( )**. Incorrect punctuation, including brace mismanagement, is a common type of mistake made by beginning C language programmers.

a) correct	b) incorrect	c) incorrect
main( )	main( )	main( )
{	{	{ .....
{	{	{ ....
{ ...	{ ...	}
...	...	}
} .	}	{ .....
....	....	{ ....
}	}	{ ....
}		}

Figure 2.2. Nesting braces

Names

Assigned variable names must follow specific rules. These names must start with a letter and may be followed by any number of letters, numerals, or underscores. However, depending on the compiler, only the first several characters are significant. (Unix as well as many compilers set this limit at eight.) Consequently **anthropod_one** and **arthropod_two** are, to compilers having a maximum name length of ten or less, equivalent. Capital letters and lowercase are treated separately. Thus **Arthropod, arthropod,** and **ARTHROPOD** are distinct names. Keywords may not be used as names. **auto1** is not equivalent to the reserved keyword **auto,** though **unsigned_1** would be equivalent to the keyword **unsigned** in a system limited to eight significant characters. Such usage, in any case, is discouraged because it leads to confusing code.

Variable names are usually selected with some mnemonic characterstic in mind. A name such as **sum** is more easily recalled than **x**. The underscore can be useful in improving variable readability. For example, **num_of_1s** would be preferable to **numof1s**. The extra effort and time employed by a programmer in creating meaningful names will usually be well rewarded during future program maintenance.

The Standard C Library

Earlier, we mentioned that all the programs in the first five chapters will contain only the single user-supplied function main(). It follows then that printf() need never be coded by the user. One might wonder how a function not appearing in a program could be called. In fact, it does appear in object code, although not in the source code. This mysterious guest appearance is due, in part, to the *standard C library*.

The C language includes no facilities for input/output, file handling, as well as many numerical, string, and other operations. One might think it strange that a language includes none of these seemingly standard routines, until one remembers that C was created to be a systems language. Depending on the application, many of these routines would be superfluous. So, instead of adding complexity to a very concise language, a library of software routines was substituted. When a program demands the use of one or more of these, then ideally, only the necessary routines will be added to the compiled program. These routines are, in fact, only normal C functions and do not structurally or operationally differ from user-defined functions. The operation and names of C library routines, like C itself, is highly portable due to the standardization introduced by Unix.

The printf( ) Function

The **printf( )** function is one of the more widely used applications functions for C. As mentioned earlier, printf()'s purpose is to output its argument list to the standard output device, usually the monitor. Because it will be used frequently in this text, a short description is in order. printf() outputs both literals and variable values. Except for *conversion specifications, escape sequences,* and double quotes, all characters within a pair of double quotes will be treated as a string constant and displayed.

✗ Program 2.1

CONVERSION SPECIFICATION

Conversion specifications always begin with the percent sign immediately followed by one or more *conversion characters*. In Program 2.1, the only conversion character used is %c.

printf(".t%c%c%c %c%",a,d,b,b,c);

This conversion character instructs the system to output the corresponding matched variable from the ensuing argument list as a character.

The %c's specify that binary data is to be converted to character values for output. The **printf( )** routine substitutes the **nth** variable from the subsequent argument list for the **nth** conversion character, where **n** represents the ordinal position of both the conversion character and matching variable. For example, in the above statement from Program 1.1, the value of variable **a** (which has been assigned the ASCII code for T or 84) is output in lieu of the first %c (the %c immediately following the \t). If one had specified the conversion character for decimal numeric output, %d, then the numeric value of **a**, which is 84, would have been output. The other conversion characters are sequentially matched in a similar fashion. Although such generous conversion character switching (i.e. from %c to %d) gave reasonable results in this case, in general, it will lead to erroneous output.

Notice that the blank space after the third %c is translated to the output of Program 2.1. While literals, such as this blankspace may be printed using several methods, variables may only be output in conjunction with conversion characters when using printf(). This is shown in the following example:

```
int f;
f = 53;
printf(f);
printf("%d",f);
```

The first **printf** statement will not output the value of **f**. It will instead produce either nothing or garbage depending on the compiler and library. The second **printf** function call is correct; it will output 53.

ESCAPE SEQUENCES

The \ **n** in line 12 and \ **t** in line 13 of Program 2.1 are examples of escape sequences. Escape sequences allow partial control over the format of the output. Here, they specify a newline (formfeed and return) and a tab, respectively. Although escape sequences are often represented by a backslash followed by another character, they are in fact only single character constants (i.e. they are stored, manipulated, and output as a single character).

Because the initial string in lines 12 and 13 direct the placement of the ensuing variable list and the format of the output via conversion characters and escape sequences, such strings are referred to as *control strings*.

Function Calls [Advanced]

A function call instruction is comprised of two parts: the *function identifier* and the *actual agument list*. This is almost identical to the construction of function headers as discussed so far.

Remember that the purpose of a function call statement is to temporarily transfer control, along with the required information to the function named in the call. By coding **dawg(5,'d',a)**, one is instructing the computer to temporarily transfer control to dawg(). Upon examining the function dawg(), one should find that its argument list is comprised of three variables. The actual arguments — **5**, the character **d**, and the variable **a** are matched to the formal arguments of dawg —for instance, **var1**, **var2**, and **var3** respectively. A visual representation of this process is provided in figure 2.3.

Function Call   **Called Function**
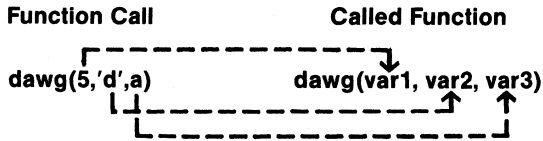
Figure 2.3 Function call argument matching

This is the method generally used for transferring information between functions in C. Now that your curiosity has been whetted, you'll be happy to know that much more will be said on this subject in chapter 6.

C Conventions

WHITESPACE USAGE

C is a freeform language. C source code must, of course, be syntactically and ordinally correct. But beyond this requirement, code format is highly flexible. This aspect can be best illustrated using a graphic example. In figure 2.4, Program 2.1 (minus the comments) has been rewritten in the most compact form possible.

```
main(){char a,b,d,c;a='T';d='h';b='e';c='n';
printf("\n Program 2.1 prints-");printf
("\t%c%c%c %c%c",a,d,b,b,c);a='d';printf
("%c",a);}
```

Figure 2.4 Program 2.1 (rewritten in compact form)

Notice that all unnecessary whitespace (blank spaces) has been removed from Program 2.1. C only requires blank spaces in the following instances:

- When used as a character value within single quotes
- Within double quotes as part of a character string
- When used to separate names, constants, and keywords (as in **goto label1**)

Generally whitespace can be added as desired in a C program. However, blank spaces cannot be inserted arbitrarily into the following:

- Names, including label, variable, and function identifiers.
- Conversion specifications
- Escape sequences
- Reserved keywords
- Strings and character values (without changing their values)
- Numeric values

Input tabs, formfeeds, and carriage returns are also normally ignored. Because of this fact, it is possible to employ run-on "sentence" structure as shown in figure 2.4.

A programming style that is overly compact is undesireable, as such a style impedes readability. This type of style can also obscure comprehension, divesting some concepts of much meaning (e.g. "single-line statements" and "structured programming"). As a rule of thumb, each statement should be placed on a separate line, although related, short statements may be grouped on one line as shown in the following example:

a=16;   c=5;   d=6;   b=a+1;

Multiple statement lines should be used judiciously.

POSITIONING AND INDENTATION OF BRACES

Since C is a freeform language, C programmers should devise and adhere to structured statement formatting. The

following paragraphs describe the method used in this text. Many variations exist.

The function identifier will be positioned to the far left and will be preceded and followed by a blank line. The function delimiting opening brace (i.e. the first {) will be placed below and in line with the **m** of **main**. Two blank spaces will separate this brace from subsequent code. Each of the subsequent opening braces, if any, will be indented one tab relative to its predecessor. All matching closing braces , } , will be aligned underneath their counterparts. Neither code nor a semi-colon will be placed on a line with a closing brace. Figures 2.2a and 2.1 demonstrate proper formatting technique.

FUNCTIONS

In order to aid the reader in differentiating called functions from other C expressions using a similar syntax, an arbitrary convention has been adopted in this text — a blank space will not be inserted between a function identifier and the following parentheses (of the argument). Conversely, all other expressions with similar structures will carry such whitespace.

CAPITALIZATION

Unlike BASIC, the bulk of C code is written in lowercase. This practice is often more than convention. Many C compilers will not accept keywords, main(), and library calls coded using uppercase characters. By convention, uppercase names are reserved for symbolic constants.

Source File and Compilation

Before one can compile or run a program, it is necessary to first place the program in a form acceptable as input to the computer system. Older systems make heavy use of punched computer cards that are read in a batch manner by a card

reader. Today most large systems and all smaller computers receive input from magnetic secondary storage such as hard disks, reel or cartridge tape, floppy disks, or newer bubble memory units.

Regardless of which type is used, the source program must be placed into a *file*, usually by itself. Most operating systems have a built-in file creation and editing system. File names are limited to a specified number of characters, although portability suggests this number be kept to eight or less. Most C compilers require that a source file have an *extension* of **c** as in **prog1.c** .

Once the source code has been placed on file and the file is properly closed and stored, it may then be compiled. Compilation procedures differ significantly among compilers, although with most, a command word followed by the source file name without the **.c** extension must be input at the operating system level. For example if the compiler execution word is **compile**, the aforementioned file could be compiled with the following line fed to the operating system:

compile prog1

Many compilers for small computers are multi-pass compilers; several separate compile steps must be executed in the correct order during compilation.

Regardless of the exact command(s) necessary to perform the compilation, both the source file (and any intermediate files for multi-pass compilers) and compiler files must reside either in main memory or in the secondary storage device designated in the compile command line or implicitly assumed in the compiler software. For detailed information on file creation and C compilation, the reader must consult system software (user) documentation on the respective topics. Additional aspects concerning compilation are touched upon throughout the rest of this text, notably in chapters 7, 8, and 11.

Review

A labelled version of Program 2.1 is presented in figure 2.5. Re-examine the format, components, and syntax of the program and be certain you understand how Program 2.1 operates. A summarization of important C features considered in this chapter follows.

Composition

- The C language, proper, is composed of seven types of codes: constants, variables, operators, punctuation, keywords, function identifiers, and labels. Comments are also allowed.
- Each variable must be declared to be of a data type and must be assigned a value (initialized) before it can be used.
- All of C's operators consist of non-alphanumeric symbols except for sizeof.
- Punctuation is used to delimit and identify specific groups of code (e.g. blocks), or simpler specific statement components or expressions.
- There are twenty eight keywords in C. All of them except for entry and sizeof serve a control flow or declarative purpose. sizeof is an operator while entry is reserved for future implementation.
- C source code consists almost entirely of user-defined functions. C library called routines are added to the object code.
- Called functions are employed much like subroutines are in other languages. The calling function transfers control to the called function, where (hopefully) the desired manipulation is performed. At the finish of the called function, control is transferred back to the calling function.

Statements

- Code must be combined in a smooth, continuous way to form statements. There are four categories of statements: declarations, function calls, operationals (assignment), and control flow (conditionals and loops).
- Statements contain the executable code of C programs.
- Every simple statement must end in a semicolon.

Functions

- Functions are divided into two sections: the function header and the function body.
- The function header is composed of two parts: the function identifier and the parenthesized formal argument list.
- The function body must be delimited by a matched pair of braces.
- Every program must have one and only one main( ).
- Libraries, such as the standard C library, contain many useful routines which can be called but do not appear explicitly in a program's source code listing.

Conventions

- C is a freeform language. Therefore it is highly advantageous to impose a clear, unambiguous format style on one's code.
- Uppercase is reserved for names of symbolic constants.

```
/* Program 2.1 */
/* This program prints several characters and a char-
acter string to the standard output device */


main()                    /* line #5 */


{
   char a, b, d, c;
```

Program identifier and description comments.

Line identifier comment

Function identifier

Formal argument (null list)

Function header:

Beginning of main( )'s body

Variable declaration statement

Variable assignment (initialization) statements

Line identifier comment

Function identifier

Function argument

C standard library printf( ) function call statements

Function argument list

(Re) Assignment statement

Line identifier comment

Another printf( ) call statement

End of main( )'s body

Program output

```
a = 'T';
d = 'h';
b = 'e';
c = 'n';

                    /* line #10 */

printf("\n Program 2.1 prints-");
printf("\t%c%c%c %c%c%c",a,d,b,b,c);

a = 'd';            /* line #15 */

printf("%c",a);

}
```

```
Program 2.1 prints-    The end
```

3

Fundamental Data Types

Introduction

Data is one of the most commonly used terms in programming. Data can be defined as the raw information input into the computer. In C, data has a more specific meaning. It refers to the values associated with constants (literals) and those values represented by variables. The two major aspects of data in C are the data type and the storage class. Fundamental data types are discussed in this chapter while storage classes will be detailed in chapter 7.

We have already introduced one data type in chapter 2 —the character. There are, in addition, three numeric data types that comprise the fundamental or primary data types of C. It is necessary that we declare each variable to hold or to be of a certain data type before using that variable. This is not the case with some other high level languages which will automatically assign variables to a class on the basis of their structure or content.

Figure 3.1 includes a schematic of the four fundamental data types and their relative hierarchical positions. Notice the data types are grouped into two sets. The lowest two types are said to be integral or to possess integer flavor, while the top two are of floating point flavor. While the proper names of the four data types are given, the associated C keywords are presented in parentheses.

```
Floating      ⎧   double floating point (double)        HIGHEST
  Point       ⎨                                             ↑
  Flavor      ⎩         ↑                                    |
                  single precision floating point (float)    |
                        ↑                                     |
Integer       ⎧   integer (int)                              |
  Flavor      ⎨                                               |
              ⎩         ↑                                     |
                  character (char)                        LOWEST
```
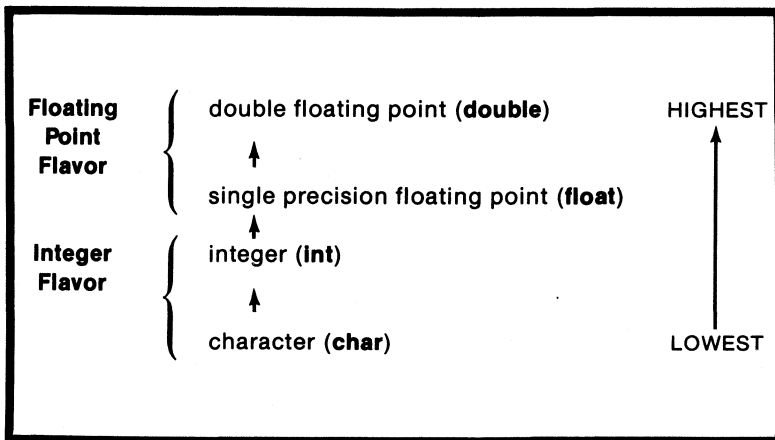
Figure 3.1.   The hierarchical positions of the four
fundamental data types

These data types are said to be hierarchical because any lower type can be [approximately] represented in the form of one of the higher types, while the converse is generally not true.

Characters

A character is a letter, numeral, or special printing or nonprinting symbol which can be handled by the computer system. These available symbols define the system's *character set*. Since computers may only handle binary data, it is necessary to devise a code to represent the character set. By far the most popular of all coding systems for smaller computers is the American Standard Code for Information Interchange or ASCII (pronounced as'key; appendix B)*. The coding process takes place at the user/input interface, whether it be a console or a card punch.

As a case in point, the numeric symbol 7 is encoded as 00110111 binary (decimal 55). When this data is input, the computer is faced with the choice of treating this byte as the character 7 (by leaving it coded as 00110111) or treating it as the value 00000111 (decimal 7). The binary numbering system and a common method of implementing it on computers is discussed in appendix C.

Such duplicity necessitates that the programmer inform the computer that this sting of bits is to be used to symbolize the character 7. *A character constant value is differentiated from any possible corresponding numeric value by enclosing it in an immediate pair of right-sloping single quotes, (e.g. '7').* Several characters enclosed in double quotes constitute a character string (chapter 9).

Variables that handle character values should be declared as **char** type before usage, as we did in Program 1.1. Initialization of such variables may be accomplished in a number of ways.

---

* Although a standard ASCII code is often alluded to, in reality ASCII code may vary from system to system. Fortunately, most of this deviation centers around special control characters that either do not concern the average programmer or can be replaced in C by appropriate symbolic escape sequences. All extended ASCII symbols (i.e. those represented by values over decimal 127) are extremely system dependent.

```
char a,b;
a = '%';
b = 37;
```

Both **a** and **b** are assigned the value of the percent character. Although **b** is assigned this value directly, one may run into a readability problem. For example, someone retyping this code section for a non-ASCII system would probably not know that 37 symbolized % and should therefore be recoded. Thus, direct ASCII numeric input also introduces a portability problem.

ESCAPE SEQUENCES

Certain special characters cannot be entered because of their nonprinting nature. Again direct numeric ASCII value assignment of variables (or similar representation using the **#define** keyword — chapter 8) can be used. Nonprinting and special characters, as well as normal printing characters can alternately be symbolized by the use of *escape sequences*. Escape sequences are specific tokens in C that begin with a backslash, \, the escape sequence lead-in character. The backslash is immediately followed by one of a restricted list of characters or by an up to a three digit octal number. Variables cannot be used in escape sequences. Escape sequences are also used to represent "difficult" characters, whose direct coding would cause punctuation or syntax ambiguities.

Escape Sequence	Represented Character
\ b	backspace
\ r	carriage return
\ f	form feed
\ n	newline
\ t	tab
\ $12_{opt}3_{opt}$	octal character value
\ '	single quote
\ "	double quote
\\	backslash

opt-OPTIONAL

Figure 3.2. C's escape sequences

All escape sequences, except those utilizing octal code, are symbolic constants inherent to the C language. The backslash-octal number combination directly supplies machine set code. (For example, ' \126' is equivalent to 'V', or decimal **86** in ASCII.) All escape sequences may only be used within single or double quotes.

The lower three escape sequences in figure 3.2 were implemented to differentiate between the use of their represented characters as character values and the use of these same characters in a special control or punctuation capacity. For example, to assign the "difficult" single quote character to the variable **sgqu**, the following statement could be used:

sgqu = ' \";

If the backslash had not been included, then the second quote would have been interpreted as the closing quote of a character value expression. In such an instance, the enclosed character would have been the null symbol (ASCII value 0). The third single quote would have been interpreted as being superfluous and would have initiated an error message.

Therefore, the \' escape sequence is used inside single quotes to represent the single quote character value (ASCII 39), while the \" escape sequence is used inside double quotes to represent the double quote character value (ASCII 34) as an element in a string.

Just as the appropriate escape sequence must be used to contrast quote character values from their respective usages as punctuation, so must an appropriate escape sequence be used to separate the backslash character from the backslash as used as the escape sequence lead-in character. The escape sequence consisting of the lead-in backslash followed by the backslash character is used for this purpose. Program 3.1, listed in figure 3.3, should clarify these concepts.

```
                /* Program 3.1 */
/* This program illustrates several ways of outputting the
       backslash character utilizing printf() */

main()                                                      /* #5 */

{ char BSL_1;
  char BSL_2;
  char PERC;
  PERC = 37;
  BSL_1 = '\\';                                             /* #10 */
  BSL_2 = 92;
  printf("\n\nThere are a number of ways of printing a backslash.");
  printf("\nOne can use the escape seqence \\\ :");
  printf("\n\tTYPING:  printf(\"\\\\\\\")  OUTPUTS: \\\");

                                                            /* #15 */
  printf("\n\nor one can use the backslash-octal ");
  printf("number combination \\144 :");
  printf("\n\tTYPING:  printf(\"\\144\");    OUTPUTS:  \\");
                                                            /* 19 */
  printf("\n\nor a symbolic constant directly assigned the ");
  printf("decimal ASCII \nvalue of the backslash character:");
  printf("\n\tTYPING:  printf(\"%cc\",BSL_2); ",PERC );
  printf(" OUTPUTS:  %c",BSL_2);

  printf("\n\nfinally one can use a symbolic constant ");   /* #25 */
  printf("\nassigned the character \\ :");
  printf("\n\tTYPING:  printf(\"%cc\",BSL_1);    OUTPUTS:   %c",PERC,BSL_1);
}
```

Program output on next page

144 should be 134


```
There are a number of ways of printing a backslash.
One can use the escape seqence \\ :
    TYPING:  printf("\\"); OUTPUTS: \

or one can use the backslash-octal number combination \144 :
    TYPING:  printf("\144");    OUTPUTS: \

or a symbolic constant directly assigned the decimal ASCII
value of the backslash character:
    TYPING:  printf("%c",BSL_2);  OUTPUTS: \

finally one can use a symbolic constant
assigned the character \ :
    TYPING:  printf("%c",BSL_1);   OUTPUTS: \
```

Figure 3.3. Program 3.1 summarizes the basic methods of outputting the backslash character.

PERFORMING OPERATIONS ON CHARACTERS

In C, the character class is handled as a type of integer. In fact, most compilers will convert the char to int data type for the purposes of argument passing and mathematical manipulation. The char data type is said to be of integer flavor because character values are necessarily stored in binary integer form. With this fact in mind, the developers of C did not impose artificial restrictions on the set of operations which may be performed on character values.

For example, an expression such as:

$$97 + 35 - 63 = 69$$
'a' + '#' - '?'

is perfectly legal in C. This expression evaluates, in ASCII, to a value of 69, which in ASCII is the character E. (Throughout the rest of this book it will be implicitly assumed that the machine set will be encoded in ASCII.) Therefore the char value of E can be assigned to a variable through the use of a statement such as:

ltr = 'a' + '#' - '?';

Generally complex character expressions are infrequently encountered because they intuitively make little sense. One common exception to this assertion concerns alphabetic characters. Upper and lower case characters are separated by a value of 32. Hence a statement of the following form can be used to obtain the lowercase alphabetic char value from its corresponding uppercase value:

s_ltr = c_ltr + 32;

Another common use of char values is as operands in comparison expressions. One can test whether the char value of one letter is greater than, less than, or equal to a second value. One very common kind of program, called a *sort* pro-

gram, depends on the ability to compare the corresponding character value of different words. Although the attributes of C necessary to code such a program have not yet been introduced, the logic of sort comparisons is outlined in figure 3.4. The second version is termed "homogenous lowercase sort because all uppercase letters have been converted to their corresponding lowercase.

Non-homogenous Character Case Sort

Base List	Decimal ASCII Representation	Sorted List
Brend	66 114 101 110 100	ADDLEMAN
Addleman	65 100 100 108 101 109 97 110	Addleman
ADDLEMAN	65 68 68 76 69 77 65 78	Brend
add	97 100 100	add
addle	97 100 100 108 101	addle

Homogenous Lowercase Sort

Base List	Decimal ASCII Representation	Homogenized Lowercase ASCII Representation	Sorted List
Brend	66 114 101 110 100	98 114 101 110 100 97	add
Addleman	65 100 100 108 101 109 97 110	97 100 100 108 101 109 97 110	addle
ADDLEMAN	65 68 68 76 69 77 65 78	97 100 100 108 101 109 97 110	addleman ⎫ ambiguous sort order
add	97 100 100	97 100 100	addleman ⎬
addle	97 100 100 108 101	97 100 100 108 101	brenda

Figure 3.4. Non-homogenous and homogenous character case sets

Note that provisions must be made for comparing a character string to one that is, in effect, a concatenation of it with another (e.g. **add**, **addle**, and **addleman**). Also, duplicate character string comparisons must be logically dealt with. While such duplication is possible under both sorting routines, it is more likely to occur with homogenization routines, as our "addleman" example illustrates.

Integers

Integers are postive or negative whole numbers. Negative integers must be preceded with the *unary minus sign*, as in **-5**. Positive integers (as well as other positive data types) may not include the plus sign **+**, as this symbol is exclusively reserved for addition. Inclusion of commas in any numeric quantity such as *12,350*, is also not allowed. Declaration and assignment of integers is straightforward.

int a;
a = 150;

The first statement declares **a** to be of integer type. **int** like **char** is a keyword in C. As a keyword, **int** must not be changed and must be delimited from subsequent text with a trailing blank space. **inta**; and **integer a**; will result in a error message such as "undefined symbol". The second statement assigns the decimal value, one hundred and fifty, to the integer variable **a**. The two blank spaces included in the second statement are optional.

Floating Point Numbers

A floating point number consists of an exponential and a fractional portion. Its value is defined as the fractional part, which is in some predetermined base, multiplied by that base raised to the exponential power. The standard base is ten while the *default* exponent value is zero. A default value is the value automatically assigned to an expression in the absence of an explicit value.

Floating point numbers take the following form, where *d* represents a decimal value:

$$.ddddde \pm dd$$

The value to the left of the character **e** denotes the fractional part (*mantissa*) while that to the right denotes the exponent. The **e** (or **E**) is used, by the compiler, to separate and identify these two components. The exponent must have a positive or negative value with the default value being positive. Table 3.1 includes several means of inputting a five significant digit estimate of pi.

Table 3.1.   Floating point input format

Input Format	Interpretation
3.1415	3.1415
3.1415e	3.1415 x 10exp0 (10^0)
.31415e1	.31415 x 10exp+1 (10^{+1})
.31415e+1	.31415 x 10exp+1 (10^{+1})
.031415e02	.031415 x 10exp+2 (10^{+2})
31.415e-1	31.415 x 10exp-1 (10^{-1})
31415e-04	31415. x 10exp-4 (10^{-4})
31415 e -4	31415. x 10exp-4 (10^{-4})

Note that in the first example the exponential portion was not included. The exponential portion's default value is base ten to the zeroth power. The exponent 10^n should be interpreted as one times n number of tens. Another way of visualizing this is as one followed by n zeroes. Any number raised to the zeroth power equals one. The default value of the exponent is zero, and therefore the default value of the exponential portion is one.

Regardless of the input format used, values are stored internally using the same format (i.e. fractional and exponential parts).

Some compilers will not allow a dangling **e** as shown in the second example. Others allow insertion of white space around the **e** as shown in the last example. To assure port-

ability, the programmer should write floating point numbers without dangling **e**'s and whitespace.

Floating point numbers are declared using the **float** keyword. For example:

> **float PI,rate;**
> **PI = .3141593e01;**

Although floating point numbers are used in many mathematical expressions due to their accuracy and range, it is important to remember that in most instances *floating point numbers are only approximations of actual values*. The reasons for this are as follows:

- Roundoff and truncation errors can occur when assigning a real value to a (shorter) fixed area of storage.
- Some numeric values cannot be represented as a finite binary fraction. (i.e. as 1/3 equals .3333... decimal)

These two aspects of floating point numbers will be discussed later in this chapter, in the section entitled "Numeric Representation". In C, float's are automatically converted to double's during operations and argument passing to functions.

Double Precision Floating Point Numbers

Often referred to as either "double precision" or "double", the double precision floating point data type is merely an extension of the floating point data type. The double precision data type normally is allocated twice as much memory storage as single precision floating point. Although the details can vary significantly among systems, double precision floating point implementations often allow approximately twice as many significant digits as the float data type. In addition, the exponent range may be increased, sometimes on the order of a magnitude.

There is, however, no automatic guarantee that calculations performed with **double**'s will be twice as accurate as the same one performed using **float**'s. Whether or not this expectation is realized depends upon the aspects of the problem being solved and the actual system representation of these two data types.

Double precision variables are declared with the keyword **double** and initialized in much the same way as **float**'s (except for possible additional significant digits in the mantissa and exponent).

```
double PI,rate;
PI = .3141592654 e1;
```

The Enumerated Data Class [Advanced]

One of the recent additions to the C language that is available on some compilers is the *enumerated data class.* * The term class is used because this category is actually composed of an arbitrary number of similar user-defined data types. Specifically, the user must specify the exact values that an enumerated data type is allowed to hold. These specified values are symbolic in nature, although the programmer has the option of explicitly specifying the associated integer values. An enumerated data type definition takes the following form:

enum *type-identifier* { *set list* }

where the set list has the following form:

{ *value-identifier* = opt *integer* opt , . . .
. . . , *value-identifier* = opt *integer* opt}

---

* The enumerated data class, along with the ability to pass copies of entire structures as arguments (chapter 10) was first implemented on Version 7 of the Unix Operating System®.

After such an enumerated data type definition has occurred, variables can be declared to be of this new data type through a declaration of the form:

enum *type-identifier var-identifer,var-identifier, . . . ;*

Enumerated variables are allowed only to take on the symbolic values of the set list of the corresponding type definition.

An example should clarify this concept. Consider the following enumerated type definition:

enum position {low, middle, high};

This statement defines **position** to be an enumerated data type with allowable values of **low, middle** and **high**. Now it is possible to define variables of data type position, as in the statement:

enum position p_tank, p_plane, p_submrn;

The three variables **p_tank, p_plane,** and **p_submrn** once declared as position's can only take on the values **low, middle,** and **high**. Hence assignments of the following types are valid:

p_submrn = low;
p_tank = p_submrn;

The first assigns the enumerated variable **p_submrn** a value of **low**, while the second statement assigns to the enumerated variable **p_tank** the value of variable **p_submrn**, which is **low** from the previous statement. The statement:

p_plane = vrtcal;

is illegal because **vrtcal** has not been declared as a value in the set list associated with the definition of data type position.

The compiler actually associates an int constant with each enumerated constant. For example, in the data type definition,

enum position {low, middle, high};

the compiler assigns the enumerated constant **low** a value of int 0, **middle** is assigned a value of 1, and **high** a value of 2. Each list, by default, has its members numbered sequentially from zero onward. The programmer can explicitly associate integer values during data type definition. Consider the definition:

**enum position {down, low = 5, middle = 10,
high = 15, orbit = 20, out};**

The constant **down** is assigned zero by default, **low** through **orbit** are assigned the stated integer values, and **out** is assigned a default value of one more than the previous constant, or 21. The associated integer values must be unique within any one set list. With reference to this last data type definition, the following statements are valid:

**p_tank = orbit;
p_plane = p_tank - high;**

The variable **p_plane** is assigned a value of **low**. (i.e. **orbit** (20) -**high** (15) = **low** (5)). This statement would result in an error if the resulting integer value was not associated with a set member.

It is permissible to combine the data type definition and the related variable definition(s) in one statement, as in:

**enum position {down, low = 5, middle = 10, high = 15,
high = 15, orbit = 20, out}   p_tank, p_plane,
p_submrn;**

The data type name, here **position**, may be optionally omitted; the enumerated data type is said to be unnamed. Only those

variables present in the definition statement of an unnamed enumerated data type may be declared as that specific unnamed type.

Unlike the other fundamental data types, there is no automatic conversion to or from the enumerated data type and other types (even other enumerated data types). Hence, the following statements are illegal:

```
p_plane = 15;        /*illegal*/
if (10 > p_plane)        /*illegal*/
    object statement;
```

The motive behind this segregation deals with one of the main strengths of the enumerated type-namely safety. It is more difficult to accidentally assign a variable the wrong value if proper mnemonics are used. In addition, the compiler thoroughly checks that only defined values are held by each enumerated variable throughout the program. On a related note, the clarity and readability of a program should inevitably be improved by proper mnemonics.

There are two last points about enumerated data types that are mentioned next because this data class will be avoided in the rest of this text. First, these data types behave much like other fundamental data types with regard to scope, longevity, and identifier uniqueness considerations (chapter 7). Secondly data type conversions can be coerced with the use of the cast operator (chapter 4). For example, in relation to the last named definition, the following statement is valid

```
p_plane = middle - (enum position) 5;
```

because the cast converts the integer 5 into its equivalent position value of **low**. Of course casting an enumerated constant or variable-held value to another fundamental data type value is always possible. Consequently the following statement is now legal:

if (10 > (int) p_plane)
object statement;

One last comment: although the enumerated data type is now a standard C feature, it is not supported by many compilers due to its recent inclusion in C.

The typedef Declarator [Advanced]

typedef is a convenient "command" in that it allows shorthand declaration. typedef is not used to declare variables. Instead, it is used to define a model that can later be used to declare actual variables. An example follows:

typedef int UNITS;
UNITS mops, disin, sponges;

In the first lines, UNITS is type defined to be an integer declarator model for variables. It is common practice to put type defined models in uppercase, since they are considered symbolic constants of a declarator type (e.g. **UNITS** symbolizes **int**). In line 2, the variables **mops**, **disin**, and **sponges** are declared to be of an integer data type through the use of the type defined model **UNITS**.

There are several reasons for using typedef's in a program. First of all, their usage improves program documentation. In the preceding example, the actual quantities of **mops**, disinfectant (**disin**), and **sponges** were declared as integers under the descriptive model of **UNITS**. If this program section required integer variables to describe a logically different set of actual data (e.g. ages and time periods), a different typedef could be used.

typedef int UNITS;
typedef int TIMES;
UNITS mops, disin, sponges;
TIMES age, per_rec, per_used;

Type definitions are also useful in improving portability. If the preceding section of code was moved to a system whose integer storage allocation was only half as great (i.e. integer numeric limits are greatly reduced), there might be a need to change all int to long int declarators in order to reserve the same number of bytes per variable. Alternately, one could type define all integers using a model. One would only need to re-type define the model to solve this portability problem.

typedef int INT_32b; ⇨ **typedef long int INT_32b;**

Type definitions are probably used most often to clarify and simplify declarations:

typedef char **ARY_P_CP[10];
ARY_P_CP sent, par, punc;

This typedef defines **ARY_P_CP** to be a model for declaring a ten-by-one array of pointers. Each pointer element in turn points to a character pointer. (The asterisk is pointer notation, while the brackets denote an array.). This operation could have been accomplished in a number of steps, since type definitions can be nested.

typedef char *CPTR;
typedef CPTR *P_CP
typedef P_CP ARY_P_CP[10];

The first line defines **CPTR** as a pointer to character values. The second defines **P_CP** as a pointer to character pointer. Finally, the third line specifies **ARY_P_CP** as a 10x1 array of element type **P_CP** (pointer to character pointers). The C programmer should concern him- or herself with the logic rather than the details of this example. Keep the following points in minds:

- The model declarator has no significance until it is typedef'd.
- Only one undefined model declarator may appear per typedef. A previously defined model will immediately follow the typedef keyword in nested type definitions.

Type definitions are handled not by the preprocessor (chapter 8), but by the compiler proper.

Data Extensions

There are several *extensions* or *qualifiers* that can be used in conjunction with the fundamental data types to create new, derivative data types. Not all of these derived data types may be available on any given system. The three extensions are **short, unsigned,** and **long**.

Each of these extensions ostensibly affects the memory allocation assigned to variables declared using these extensions. The short and long extensions were designed to change the number of bytes allocated to store the value of the variable. Obviously, any change in the amount of memory reserved for a variable will change the maximum and/or minimum values the variable can hold. These boundary values are called the *upper* and *lower numeric limits* respectively.

The long extension increases this allocation and thereby allows larger absolute values to be assigned to selected data types. On the other hand, short was designed to decrease the storage allocation and the limits of the stored data. The extension unsigned suppresses the one bit sign extension, thereby providing an extra bit for use in numeric representation. Although the use of the unsigned extension doubles the upper or positive numeric limit, it contracts the lower or negative limit to zero. Therefore, the difference of the upper minus the lower limit, termed the numeric range, remains unchanged with the unsigned extension. The extensions long, short, and

unsigned are applied to integers as illustrated by the following examples:

short int s = 250;
unsigned int a = 65000;
long int l = -820000;

Up to this point, there has been no mention of either actual memory usage or the numeric value range associated with data types. This is due to the fact that these details are *machine* or *system dependent*. That is, these details are not formally specified in C. They are left up to the hardware and software designers. Memory requirements and attendant numeric limits should be listed in compiler documentation.

Table 3.2 lists the fundamental data types as well as extension-derived data types in ascending order of *widening* hierarchy. The adjective "widening" refers to the general increase in memory usage associated with "higher" data types. Like the hierarchy in figure 3.1, any of the data types can be (approximately) substituted for with one of the higher types. This table also contains memory usage and numeric ranges typical of the current generation of micro and small minicomputers.

Notice in table 3.2 that the numeric range (i.e. upper minus lower numeric limit) for all of the listed data types except those of floating point flavor (i.e. **double** and **float**), is 2 exp (storage length in bits). This is due to the direct conversion of decimal numeric values to binary data. Without concerning ourselves with the full details of numeric data storage at this time, let's examine more closely the data types and extensions listed in table 3.2.

char

The character data type must, as a mimimum, allow the representation of the 128 standard ASCII (or other like EBC-

Table 3.2. The widening hierarchy and some typical
associated values.

Data Type	Storage Length in Bits	Numeric Limits in Base Ten
double	64	*approximately* $\pm$ 10 exp -307 to $\pm$ 10 exp 308
float	32	*approximately* $\pm$ 10 exp -37 to $\pm$ 10 exp 38
unsigned long int	32	0 to +4,294,967,295
long int	32	-2,147,483,648 to +2,147,483,647
unsigned int	16	0 to +32,767
int	16	-32,768 to +32,767
short int	16	-32,768 to +32,767
char	8	0 to +127 plus 128 variable
(char)	(8)	(-128 to + 127)
(unsigned char)	(8)	(0 to + 256)

DIC) character values. Sometimes there is a need for additional capacity, as some I/O devices permit usage of nonstandard character values. On machines that store char values as eight bits, there may be an additional 128 characters, stored in any of a variety of ways (see "char vs unsigned char [Advanced]). char quantities are converted to int's during most operational manipulations and function argument passing.

short int

The short extension can be applied only to the int fundamental data type. The short int data type was created with storage economy in mind. On some large computers the short int data type will be "thinner" (i.e. shorter in bit usage and

numeric range) than the int data type. However, on the vast majority of micro and mini-computers, these two types are handled identically. In any event short's are converted to int's during operations and argument passing.

int

The int fundamental data type has already been discussed. Since integers are generally the most widely used data type, it is important to be familiar with the details of the integer implementation, especially the associated numeric limits.

unsigned int

The unsigned extension allows usage of the sign bit of a memory block for numeric representation. Its use does not change the range of a data type. Instead, it shifts the numeric limits to positive values. In many applications where negative values are not encountered, this is the most efficient method of extending the effective limits of an int variable.

long int

The long extension can be used with the int and, under some compilers, the float fundamental data types. Although the implementation varies, the long extension usually doubles the amount of memory allocated with a data type. This can be useful for programs that need to accurately handle large integer values, such as population studies and polling. long int's vary roughly from plus to minus 2.1 billion. These limits are sufficient for the majority of integral applications.

unsigned long int

This declaration combines the effects of the unsigned

and **long** extensions. Variables having positive integer values up to roughly 4.2 billion may be represented using this declaration.

float and double

The general attributes of these two data types have already been discussed. Remember that data types of floating-point flavor should be thought of only as approximations of real values due to possible rounding errors.

Examine the numeric limits of these two data types. These are only approximate limits as relatively large variations exist in the way floating-point values are represented. Also, note that the exponential limits appear not to be whole powers of two. This is due to the fact that the exponential limits in table 3.2 are presented in powers of ten. The exponential base is converted to two. More will be said on this subject in the section entitled Numeric Representation [Advanced].

long float

The **long float** data type, when enabled, is almost identical in all respects except name to the **double** fundamental data type. When this is true there is little reason to use **long float**'s as this declaration is not highly portable.

char vs unsigned char [Advanced]

This section deals with the numeric representation of character values and is primarily aimed at those users who are concerned with the manipulation, transmission, and storage of character values. C makes no stipulation on this subject, other than the allocated memory block must be sufficient to hold the 128 standard characters. In all current implementations, how-

ever, characters are stored in one byte, usually comprised of eight bits. In such a scheme, the numeric char values from 0 to 127 will be stored in normal binary format, and they will occupy the first seven bits. The eighth bit is normally used as a sign bit. If one attempts to store a numeric value over 127, as would happen when an extended, non-standard character value was being stored, the eighth bit must be filled with a one. It is the interpretation of this eighth, sign bit that leads to complications.

Consider the ASCII character value decimal 254 or binary 11111110 (ignore the char to int automatic conversion). Table 3.3 shows how this memory string is interpreted as a binary number in the different binary format systems. The logic behind the two's complement interpretation of the **signed** string is given below:

1. binary string   11111110
2. subtract one   11111101
3. one's complement   00000010
4. numeric interpretation   decimal -2

Table 3.3.   Sign extension interpretation on a character value of 254 decimal.

Binary Format	Decimal Interpretation of the Binary String 11111110	
	signed	unsigned
Simple binary	-126	254
Two's complement	-2	254
One's complement	-1	254

Note that the interpretations in table 3.3 for the **signed** string differ greatly. In normal operations on strings, such as addition and subtraction, and in normal output operations this

causes no problem because internal consistency is maintained. For example, subtracting three from this value will result in different interpreted signed values under these formats. However, in all cases the resultant values represent the same character. Internal consistency breaks down, however, in many instances where other operators such as logical, multiplicative, shift, etc. are used. The interpretation of the sign bit can also cause problems during data transmission if data format conversion is necessary.

These problems can be avoided by the suppression of the sign interpretation. Some compilers will inherently do this while others explicitly enable this through the use of the **unsigned char** data type. Note that the interpretations in the unsigned column of the table are homogenous and in agreement with the original ASCII value. Program 3.2 prints out a system's character set. The character sign ambiguity was avoided by the use of an integer variable. If the output device is not fully compatible with the computer, the output of Program 3.2 may not faithfully represent the computer's machine set.

```
                /* Program 3.2 */
    /* Outputs ASCII character set (printing
       and non-printing) and associated deci-
       mal values                       */
    main()

    {   int a;
        a = 0;
        while (a <= 255)
            { printf("\t%c(%d)",a,a);
              a = a + 1;
            }
    }
```

Figure 3.5. Program that outputs a system's character or machine set

You may already have asked yourself why since the **char** fundamental data type can be mathematically manipulated, it can not be used in place of a short integer. This is true to a large degree. Depending on your compiler, numbers in the limits of 0 to 255 or -128 to 127 can be stored in the **char** data type. *In most systems, when a* **char** *variable is mathematically manipulated or passed to a function, it is converted to an* **int** *value first.* Therefore, the savings in memory using this scheme is somewhat less than the ratio of memory allocations for **char** to **int** would suggest.

REMARKS ON WIDENING HIERARCHY [Advanced]

One should keep in mind that storage length, numeric limits, and even the allowed data types are system dependent. One should not rely on possible system compatibility when writing portable programs.

As a precaution, one should avoid the non-zero numeric limits of data types, especially in regard to floating point flavor data types. Variables and constants that are within limits on one system may be out of range on another. It may be necessary to redeclare **int** variables as **long int** when moving a program from a machine with a large *word size*, the number of bytes that are handled as a unit by the CPU, to one with a smaller word size.

System programmers often must know the memory length of a data type. However as mentioned previously, C has almost no criteria concerning this quantity.* C provides for this type of situation, and does so without compromising portability. The **sizeof** operator gives the length, in bytes, of its operand. If the operand is a variable, it must, of course, already be declared. The **sizeof** operator is described in chapter 4.

---

* This is another example where the language was designed with an eye towards generality and portability.

Not only must a programmer address machine to machine portability problems, he or she should also be concerned with the "portability" of a program over time. For example, although the information in table 3.2 is representative of the recent 8 and 16 bit internal bus mini and microcomputers, a new generation of these computers having 16 to 64 bit internal busses will probably make this information obsolete. Generally, an increase in internal bus or word size leads to an increase in data type memory allocation and a corresponding increase in numeric limits.

DEFAULT DECLARATION

C has a built-in convenience that allows the programmer to declare certain variables in a shorthand fashion. When using one of the extensions: unsigned, short, or long, the keyword int can be omitted. In other words, int is the default fundamental data type for these adjectives. Therefore, the following declarations,

```
long    lg1, lg2, lg3;
unsigned    uns_1, uns_2, uns_3;
short    shrt1, shrt2;
```

are equivalent to:

```
long int    lg1, lg2, lg3;
unsigned int    uns_1, uns_2, uns_3;
short int    shrt1, shrt2;
```

Although this shorthand method decreases coding time, it may be confusing to the beginner. C also includes a widely used default declaration for functions (chapter 6).

The Need For Different Data Types

Since all allowed data can be approximately represented in double precision floating point format, why use the lower

data types at all? The first reason can be explained by the qualifier "approximately". Floating point quantities are only approximations to real values. While the use of the double data type may only introduce an error in the order of roughly .0001%, such an error is still unacceptable in many applications. For integral calculations, long int's are much more appropriate than double's for larger values.

Secondly, floating point data types generally require more memory storage than do the integral types. Although memory efficiency is generally not paramount to the applications user, it is always a good programming practice to code efficiently.

The third reason floating point data can be inappropriate relates to compile and run times. The use of floating point numbers over integral types will slow operations for two main reasons:

- Since floating point numbers generally are stored in more bytes than integer flavored numbers, and since all but the largest of computers can only process a small number of bytes in one step, "bulky" floating point quantities must be broken down into more "pieces".
- Binary representation and mathematical manipulation of floating point values are much more involved than for integers.

Although the specifics of this second reason are beyond the scope of this book, the difficulty is related to the basic nature of a floating point number. Converting both fractional and exponential decimal values to their binary equivalents, or conversely from binary to decimal, is much more arduous than converting integer quantities. A simple analogous example would be converting .1 in base seven (i.e. the fraction 1/7) to its decimal equivalent, .14285. Also, the two interrelated parts of a floating point number, the fraction and the exponent, must be handled separately and in a different manner.

Some systems partially alleviate this problem by their inclusion of a *math co-processor*. A math co-processor is a special purpose microprocessor chip which executes specific mathematical operations and is controlled by the main microprocessor.* Even in a system employing this device, integer arithmetic operations are still executed with greater speed.

The final reason for avoiding floating point usage relates to the philosophy and intention of C. Recall that C is primarily a systems language. Hardware systems do not deal in fractions but only in integral quantities. Thus, there is rarely a need for the supporting language to deal in such quantities. Integer flavored data types are the workhorses of C.

More On Constants

Recall from chapter 1 that constants can be of non-numeric (e.g. 'h', '3', or "take 5") or numeric (e.g. **3**, **120975**, or **5.1211e05**) literal type. Character type values are enclosed in single quotes, while character strings are enclosed in double quotes (strings and arrays are discussed in chapter 8). Numeric decimal integer data is written in conventional form without the commas (e.g. **1234567**), while floating point numbers may be written in a variety of forms (e.g. **.01234**, **.1234e-1** , **.001234E-01**, etc).

The alert reader should have noticed that constants have not been declared formally, nor is this type of statement allowed. For example, coding:

int 15;

would result in an error. Only variables may be explicitly

---

* Here we have an example where the term CPU is not synonymous with microprocessor, as *two* or more microprocessors comprise the CPU. Larger computers commonly contain two or more microprocessors and related electronic components.

declared. (The compiler would, in fact, assume that **15** is an illegal variable name since it does not begin with an alphabetic character.) C allows the use of non-decimal (i.e. non-base ten) numbers. These numbers must be identified as non-decimal. This identification is not accomplished with a declaration statement but instead through the use of prefixes.

- Decimal numeric literals begin with a non-zeo digit and consist only of digits and an optional decimal point, minus sign, or optional exponential portion. Decimal numbers may be of integer or floating point flavor.
- Octal (base eight) integers are coded with a leading zero and immediately followed by digits. No floating point values are allowed.
- Hexadecimal (base sixteen) integers are coded with the prefix zero plus an eks (**0X** or **0x**) followed immediately by one or more hexadecimal numbers **0** through **F** or **f**. No float's or double's allowed.

Observe that floating point values can only be represented in decimal. Appendix A contains a binary-octal-decimal-hexadecimal-BCD conversion table. Throughout the rest of this text, unless otherwise specified, base ten will be used. Values from negatives to positive seven are represented identically, save for the leading base identifier, in the three afore-mentioned bases (e.g. decimal five, **5**, is equivalent to octal five, **05**, and hexadecimal five, **0x5** or **0X5**).

Because all data is stored in binary form, the numeric value of the limits for integer flavor data types remains unchanged, but the numerals employed to represent these values will change with the base. For example, the upper limit for unsigned integers in table 3.2 is 65535 decimal. This same value can be represented as 1111111111111111 binary, 177777 octal, and FFFF hexadecimal. Consequently, the programmer must be careful not to inadvertently exceed the numeric limits regardless of what base he or she is operating in.

printf() enables output in decimal, octal, or hexadecimal format by the use of the %**d** (or %**f** for floats), %**o**, or %**x** conversion characters respectively. Study figure 3.6 on page 94.

Program 3.3 simply assigns values to three integer variables and a floating point variable. The integer variables have names indicative of the base in which they were assigned numeric values. Hence, **oct_n** is assigned the value of octal fourteen. The variables' names and their numeric values in the octal, decimal, and hexadecimal number systems are output in tabular form. The reader should confirm that each integer variable row contains equivalent numeric representations of the same value.

One should think of this process as a conversion of input data to base two for internal representation, followed by the conversion of this binary information to the base specified by the conversion character(s). The conversion characters %**o**, %**d**, and %**x** are used in the control strings in conjunction with escape sequences and whitespace to obtain the desired output format. Since tab placement varies among various output devices, output on the reader's device may differ slightly.

After the integer variables have been printed, the variable **flt_num** is output in floating point, and on the next line, in the previously mentioned number bases. Two aspects of this section of output are worthy of note. First of all, the value input, **45.67**, has not wholly survived the number system conversion process. This output reflects the fact the floating point representation is often imperfect. Secondly, the last line of output clearly demonstates the fallacy of liberal substitution of conversion characters. Outputting a variable possessing floating point flavor as an integer will invariably result in this type of "garbage" error.

```
            /* Program 3.3 */
/* The following program exemplifies the use of octal, decimal,
   and hexadecimal numeric values and conversion characters */

main()                     /* 5 */

{ float flt_num;
  int oct_n, dec_n, hxd_n;
  dec_n = 14;
  oct_n = 014;             /* 10 */
  hxd_n = 0x1B;
  flt_num = 45.67;
  printf("\n\t\t    Numeric Value Base");
  printf("\n\t\n\t\toctal \tdecimal   hexadecimal ");   /* 14 */
  printf("\n\noct_n \t\t %o \t  %d \t  %x",oct_n,oct_n,oct_n);
  printf("\ndec_n \t\t %o \t  %d \t  %x",dec_n,dec_n,dec_n);
  printf("\nhxd_n \t\t %o \t  %d \t  %x",hxd_n,hxd_n,hxd_n);
  printf("\n\nflt_num = %f",flt_num);
  printf("\nflt_num \t %o \t\t%d\t  %x",flt_num,flt_num,flt_num);   /* 20 */
}
```

Program output on next page

```
                    Numeric Value Base
                octal    decimal   hexadecimal

oct_n            14        12         C
dec_n            16        14         E
hxd_n            33        27        1B

flt_num = 45.669998
flt_num           0     -32768      D5C2
```

Figure 3.6. Example program 3.3 demonstration of octal, decimal, and hexadecimal conversion

Octal, decimal, and hexadecimal literal numbers exceeding the values of normal int limits are assumed to represent long values. However, C allows explicit declaration of such constants as long by immediately postfixing these constants with an upper or lower case "ell" (i.e. the character l or L). Prefixing a conversion character with an ell (e.g. %ld) indicates output should be in long format. Long format will output a variable or constant value as if it was of long int data type. The number of outputted digits may or may not reflect actual significance, depending on the details of the problem. On some compilers the long format can also be obtained by the use of a capital conversion code. For example, %D would output a value in decimal long int format.

Initialization Of Variables

COMBINING INITIALIZATION AND DECLARATION

Upon declaration, a variable is assigned a block of appropriate space in memory — more specifically in a portion of main memory referred to as the *heap*. For the simple data types presented in this chapter and for arrays and pointers, the bytes that comprise the above block are said to be *contiguous*. That is, these bytes occupy memory addresses adjacent to each other, so that they form a unbroken string. Ordinarily, this memory area will contain whatever information it held prior to being allocated to the current variable. This value cannot be known beforehand and therefore is generally useless to a programmer. Therefore, with the exception of **external** and externally declared **static** variables (chapter 7), the explicit initialization of variables is often a necessary step.* There are two ways to initialize a variable.

- By using input from devices outside the program, such as console input or input from a data file.
- By using one of the assignment operators, such as the simple assignment operator,=.

Up to this point, separate declaration and assignment statements have been used. However it is common practice to combine the declaration and initialization of variables into one statement. For example, the following two lines can be combined:

```
int a ;
a = 30 ;
```

giving   **int a = 30 ;**

---

* Actually variables that are declared and then explicitly "initialized" in a separate statement are really initialized to the previously held garbage value.

Such a procedure is allowed with all the types discussed in this chapter. While each initialization is allowed for only one variable (i.e. **int a** = **b** = **30;** is not allowed), a statement can contain more than one such initialization.

char a = **30, b** = **'b', c** = **' \", d** = **0Xd;**

Observe that commas are used to separate each assignment expression. Also, note that the assigned values may be of several forms.

It is also permissable to use an expression that can be evaluated, at the time of statement execution, to a constant. The following statement,

long int a = **0xA2B5C,b** = **a+4;**

is acceptable, while the expression,

long int b = **a+4, a** = **0xA2B5C;**

would cause an error message, such as "undefined symbol **a**". Notice in the acceptable statement that **a** was defined prior to its usage in an expression. In the unacceptable expression, **a** was not defined prior to its inclusion in the expression. With a few exceptions, a section of code cannot utilize information that is defined later in the program.

OVERFLOW AND UNDERFLOW

Although each system establishes limits for each data type, C compilers generally do not closely oversee memory usage. This is an advantage in that the programmer is allowed a great deal of freedom as to the form in which data will be stored as well as control over how new data types and existing data types are manipulated.

Unfortunately, *C contains no inherent checks against overflow and underflow conditions.* Overflow is a condition

where a value exceeds the upper limits of the data type of the variable to which it is being assigned. Underflow is a condition where the lower limit has been violated. These can be serious problems — not only in initialization statements, but especially in general operational (assignment) statements. Without an error interrupt, a program will run and can output reasonable data, yet the output values will be in error due to overflow or underflow conditions. Depending upon the system, the value will either be truncated and/or rounded in order to conform to the data type of the variable to which it is being assigned. Special utility routines have been designed which are available, either as a compiler option or as library functions that check for underflow and overflow conditions as well as unorthodox *type conversions* (q.v.).

TYPE CONVERSION [Advanced]

Variables of any specific data class can be assigned values of another data type. Earlier in this chapter, it was stated that the value of a lower data type can be (approximately) represented in the form of a higher type. One can, in fact convert an integral data type value to a wider integral data type or a floating point flavored data value to a wider floating point flavored data type without loss of significance. Thus a character value can be stored as an integer or a precision floating point value can be stored as a **double** safely.* This process of changing the data type used to store a value, from a lower ("thinner") to a higher ("wider") data type, is termed *promotional type conversion.* Promotion of an integral value to its corresponding floating value is not without risk, because low order significance may be lost due to the rounding process that occurs with floating point flavored representations.

---

* If this promoted value is not "greatly" altered during its stay in the higher data class it may be demoted or returned to the original data class without significant loss of fidelity.

The outlook is much different when one assigns a higher data type value to a data type of a relatively lower class. This process is called *demotional type conversion.* When we perform this process through an assignment operator (or through some type of input function), the value of the variable, constant, or expression on the right side of the assignment operator (or the input value) will be demoted or lessened in order to comply with the data type of the variable to the left of the operator (or variable accepting the input value). The process of demoting one integral value to a lower or thinner data type consists of truncating any out-of-range higher order (the farthest left) digits in the binary representation.

The demotion of a **double** or a **float** data type involves the rounding off of any extra lower order digits in the mantissa and what amounts to the truncation of any out of range higher order digits in the exponent. The demotion of a floating point flavor data type to an integer flavor data type will be successful if the value is within the limits of the latter type. In this instance, any fractional portion will be either truncated or rounded. If the value is not within the limits of the integer flavored data class, the exact result will depend on the compiler, although one can be certain that it will be "garbage".

Program 3.4 illustrates many of the aforementioned points in a more graphic manner. The reader should have no difficulty following the logic of this program. It simply assigns values to a number of variables and then prints these values in a loose table. One should be familiar with the escape sequences and the character conversions used. At this point, our primary concern is the process of type conversion as it occurs in this program.

Lines 10 through 14 initialize four **char** type variables with several constants of diverse flavor. Line 10 is a direct character assignment while line 11 uses the ASCII code value. Both variables **chr1** and **chr2** can be output as the character U by

using %c specification or as the ASCII value 85 by using %d. **chr3** is assigned the decimal value 341 which is binary 101010101. Since this binary number exceeds the eight bit character memory storage allocation, its extra high order bits will be truncated.* After truncation, the value 01010101 (decimal 85) remains. This value is assigned to **chr3**.

Line 14 demonstrates the demotion of a floating point number to a char value. In this instance, regardless of whether rounding or truncation of the fraction occurs, the value assigned to **chr4** is zero, or an ASCII null.

An example of the promotion of a character value is shown on line 16. Because both the variable and the literal are of integral flavor, no complications were associated with this promotion. On line 17, the fractional portion of **84.76** was truncated before its assignment to variable **int2**, leaving **84**. **int4** was assigned the value **65543** on line 18. This value exceeds the limit for int's (32767). Therefore, the high order bit was truncated leaving a value of **7**. (In base two this operation is more apparent, 1000000000000111 $\Rightarrow$ 0000000000000111). Since the value of **int4** is in range, only its fractional portion will be dropped. However, **int5** was assigned a value that was out of range, so its high order bits and fractional portion were truncated.

Now, direct your attention to the code and output associated with the variables of floating point flavor. Note that the character U (ASCII value 85) is correctly represented in both **float** and **double** data representation. The value assigned to **flt2** however exceeds the **float** variables ability to store significant digits. Therefore, the value assigned to **flt2** is only an approximation. Notice that **dbl2** was assigned and was able to correctly store this same value.

---

* There is only one extra bit.

```
/*     Program 3.4  -   Example of promotional
  and demotional data type conversions.        */

main()                                               /* #5 */

{  char  chr1, chr2, chr3, chr4 ;
   int   int1, int2, int3, int4, int5;
   float flt1, flt2, flt3, flt4;
   double dbl1, dbl2, dbl3, dbl4;                    /* #10 */

   chr1 = 'U';
   chr2 = 85;
   chr3 = 341;
   chr4 = .345e-4;                                   /* #15 */

   int1 = 'U';
   int2 = 84.76;
   int3 = 65543;
   int4 = .12345e3;
   int5 = .12345e6;                                  /* #20 */

   flt1 = 'U';
   flt2 = 1234567890;
   flt3 = .473e99;
   flt4 = 12345.678912345;                           /* #25 */
```

```c
dbl1 = 'U';
dbl2 = 1234567890;
dbl3 = .473e99;

printf("\nchr1 = %c or %d ",chr1,chr1);         /* #30 */
printf("\tchr2 = %c or %d ",chr2,chr2);
printf("\nchr3 = %c or %d ",chr3,chr3);
printf("\tchr4 = %c or %d ",chr4,chr4);
                                                 /* #35 */
printf("\n\nint1 = %d ,\tint2 = %d ",int1,int2);
printf("\nint3 = %d ,\tint4 = %d ",int3,int4);
printf("\nint5 = %d ",int5);
                                                 /* #39 */
printf("\n\nflt1 = %f ,\tflt2 = %f ",flt1,flt2);
printf("\nflt3 = %f ,\tflt4 = %f ",flt3,flt4);

printf("\n\ndbl1 = %f ,\t\t dbl2 = %f ",dbl1,dbl2);
printf("\ndbl3 = %f ,\t or %e ",dbl3,dbl3);
printf("\ndbl4 = %f ,\t or %e ",dbl4,dbl4);
                                                 /* #46 */
}
```

Program output on next page

```
chr1 = U or 85 ,          chr2 = U or 85 ,
chr3 = U or 85 ,          chr4 =    or 0 ,

int1 = 85 ,     int2 = 84 ,
int3 = 7 ,      int4 = 123 ,
int5 = 32767 ,

flt1 = 85.000000 ,        flt2 = 1234567936.000000 ,
flt3 = 34028236692093845910000000000000000000000000.000000  ,  flt4 = 12345.678711 ,

dbl1 = 85.000000 ,          dbl2 = 1234567890.000000 ,
dbl3 = 472999999999999951200000000000000000000000000000000000000000000000000000000000000000000000000000000000.000000 ,   or 4.730000E+98
dbl4 = 0.000000 ,           or 1.113038E-308
```

Figure 3.7.   Example program 3.4 demonstrates promotional and demotional type conversions

flt3 was assigned a value whose exponent was out of range. The demotion of this value to float width left "garbage", as demonstrated by the output. Again the same value was assigned to a double variable, **dbl**. The exponent was then in range, and the value was quite successfully approximated. Note that using the exponential (%e) conversion character will suppress the printing of non-significant digits. The value assigned to **flt4** was in range. However, all of its digits could not be stored significantly. Observe that the subsequent rounding operation was not performed faithfully, from a mathematical view, since **.678912345** was rounded to **.678711**.

Most languages, with C being no exception, allow variables to be declared, even though they go unused throughout the program. In Program 3.4, **dbl5** is declared but not used.

dbl4 is an uninitialized variable whose value (old memory storage contents) will be output. Obviously, this value has little significance because it is a value from a random location in the heap. Such usage may cause unexplained phenomena such as a program *abort* or system *lockout* under simple systems.

Numeric Representation [Advanced]

Appendix C summarized the basic aspects of the binary numbering system and a means of representing negative binary numbers termed two's complement. The basic components of a computer system — the memory and logic circuits — are *bistable*; that is, they are capable of taking on only one of two possible states at any one time. Bistable elements can logically only represent data employing some type of binary numbering scheme. C does not specify what internal data type representation must be used. However, the following subsections present the schemes that are very commonly employed on a wide variety of computers.

SIGNED INTEGRAL DATA TYPES

This class of data types is most commonly represented utilizing a combination of pure binary for positive values and two's complement for negative values. Accordingly the highest order binary digit is reserved for what is, in effect, a "sign digit".

Consider the char data type, which normally has a memory allocation of eight bits or one byte. Since the leftmost bit is reserved as the sign digit, seven digits remain with which to represent any given value. Therefore the limits on signed char values are 2^7-1 (i.e. 127) represented by the string:

0	1	2	3	4	5	6	7	bit number [*]
0	1	1	1	1	1	1	1	
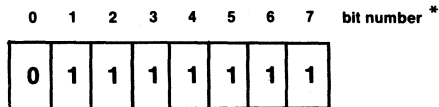
and -2^7 (-128) represented by the string:

0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0
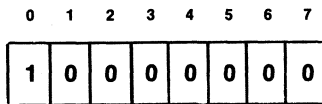
The latter is obviously in two's complement format. The areas in main storage that hold character bit strings are termed *character fields*.

This logic can easily be extended to other signed integral data types. For example, since this text's reference system stores int's in two bytes, the decimal value 4097 would be represented as:

---

[*] The bit number corresponds to the location in memory where the bit resides. Here it has been assumed for simplicity that the characters are stored in the first byte in memory. In any advent the memory is arranged so that the first bit of a byte is always a positive multiple of 8 (or zero). The general bit number form would therefore be n8+0, n8+1, ..., n8+7, where n is a positive integer.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1

-8198 would be represented as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0

One additional restriction usually associated with non-char types is that of *field alignment*. Data types that are allocated m bits usually are placed in main memory so that their leftmost bit resides on a memory location that is an integral multiple of m. For example, if an integer is stored in 16 bits, then an integer can only begin at memory locations 0, 16, 32, ... Actually only a portion of main memory is available for storage because other objects — such as the operating system, the user's program itself, etc, — must also be present. However the concept of field alignment still holds.

UNSIGNED INTEGRAL DATA TYPES

Unsigned quantities are handled throughout as pure binary quantities. In other words the leftmost digit of the field is not interpreted as a sign digit but rather as another binary digit. For example, the field;

0	1	2	3	4	5	6	7
1	0	0	0	1	0	0	0

would be translated as -120 decimal for a (two's complement) signed digit. This corresponds, under an unsigned format, to 136. The use of the unsigned extension allows values from 0 to 2^n-1 to be represented, where n is the number of bits allocated to the unsigned data type field.

FLOATING POINT FLAVORED DATA TYPES

One common format for representing floating point numbers divides the field into three components: the sign bit, the exponent, and the mantissa. The float data type is often allocated 4 bytes and is divided as follows:

0	1 through 7	8 through 31
S	exponent	mantissa
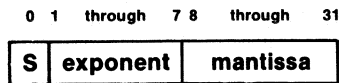
The reader should bear in mind throughout the following discussion that the binary data in the three fields is quite dissimilar and is handled in a separate manner by the computer.

Before any direct discussion of these memory fields can begin, it is essential to present just how a decimal number is transformed into its binary counterpart. For the purposes of this discussion, this process can be represented in three steps. Included is a concrete example using the decimal number .375e01.

1. Convert the value to its plain decimal equivalent by multiplying by the exponential portion.

$$.375 \times 10^1 = 3.75$$

2. Convert the decimal value to its binary equivalent. (There are a number of quick algorithms for doing this that are not presented in this text.)

$$3.75_{10} = 11.11_2 \quad \text{where the subscripts denote the base}$$

3. Put this binary number into fractional form by multiplyng by a power of sixteen.

$$11.11 = .001111 \times 16^1$$

Sixteen is 2^4, so shifts of four binary digits can be performed. At least one of the first four digits after the "decimal" point should be a one. This form is known as *normalized* format.

Now on to the components. The sign bit is again the leftmost bit in the field. A one value represents a negative mantissa value while a zero in this position signifies a positive mantissa. *Unlike integral representations, a two's complement operation is not performed on the mantissa.*

The exponent field represents the power of sixteen obtained from the decimal to binary transformation. This value is *biased* by adding 64 and is stored as a binary number in the exponent field. Consider $.001111 \times 16^1$. Since the exponent is one, the value stored will be $64_{10} + 1_{10} = 65_{10} = 1000000 1_2$. A leading one in the exponent field indicates a positive exponent while a zero implies a negative exponent value. Notice the biasing eliminates the need for any type of exponent sign bit.

The fractional part of the binary number is directly moved to the mantissa field (without the decimal point). Truncation or padding of zeroes occurs to fill the 24 bits of the mantissa field.

The original decimal value $.375e01$, which is equivalent to $.001111 \times 16^1$, may now be represented as a **float** as follows (spaces added):

sign bit $\Rightarrow$ 0

exponent $\Rightarrow$ $64 + 1 = 65_{10}$ $\Rightarrow$ 100 0001

mantissa $\Rightarrow$ .001111 $\Rightarrow$ 0011 1100 0000 etc.

Thus the field would contain the following string:

0100 0001 0011 1100 0000 0000 0000 0000

which is often converted to hexadecimal for output convenience:

413C0000

There are many deviations to this scheme, but most incorporate these same concepts. The double data type representation simply includes more digits in the mantissa and possibly the exponent fields. Field alignment also occurs for floating point data types.

One last point about floating points: many microcomputers actually do not directly allow floating point data types. That is, their CPU's are not equipped to manipulate floating point data. Instead, software routines are used to handle each component field of a floating point as an integral quantity. Obviously then, floating point manipulation on such machines will be many times slower than the corresponding integer operation.

4

Operators

Introduction

    This chapter deals with C's rich set of operators. All forty-three operators are detailed here, often with examples and special notes on their use. Operators can be classified by the number of operands they require or by the general purpose or effect of the operation they perform. In this vein, the common addition operator, +, could be described either as a *binary* operator (because it operates on *two* quantities) or as *arithmetic* operator, respectively. In this text, the latter classification scheme will be followed more closely for the sake of organization.

111

Computers, contrary to popular belief, can only perform a very limited range of activities. Specifically, there are only five basic operations which can be performed by most computers.

- A computer has the means to store and retrieve data in binary form. Despite the fact that the medium on which this binary information is stored can differ radically from one machine to another, the net effect is the same. Namely, a binary one is stored as an arbitrarily defined "on" bit while a zero is represented by an "off" bit. A computer can hold millions of bytes in main memory, with each byte normally comprised of eight bits. There are also a few memory storage blocks, termed registers, in the CPU. Peripheral memory is generally not considered to be a part of the computer, proper.

- Computers must communicate, not only with the outside world (I/O), but also with their separate component parts. The most common I/O devices are those designed for human communication such as the keyboard (input), monitor (output) and printer (output). I/O can also be of a more unconventional sort. To cite an example, industrial comptuers commonly accept environmental input and, based on this information and previous programming, output appropriate data. Here, output data can be fed to a monitor to alert industrial supervisors or can be used to directly control said process.*

  The internal components of a computer routinely communicate directly with one another or indirectly through the CPU. For example, the microprocessor often gives the memory management unit an address. The memory management unit then retrieves the binary data from this address and feeds it to the microprocessor.

---

* When used in such a direct fashion to control automated processes, the sensor(s), computers, actuators, and associated components are known as an industrial robot. Analogue-to-digital input and digital-to-analogue output signal conversion are often necessary.

- Computer circuits are capable of making simple logical decisions. These decisions deal only with the magnitude of bits or a predefined string of bits. In other words, a computer can only compare two quantities and decide whether one quantity is equal to, greater than, or less than another.

- Computers have circuits that can perform basic arithmetic operations. These circuits are often ascribed of being capable of the four basic arithmetic operations: addition, subtraction, multiplication and division.* Although certain computers such as calculators may seem to evaluate higher algebraic and trigonometric expressions, in reality, these machines actually use simple mathematical formulas, called *algorithms,* to approximate the true outcome. The specialized arithmetic and logical circuits are usually collectively termed the Arithmetic/Logic Unit, or ALU.

- The ALU also contains circuits capable of bitwise logical operations, such as the NOT (one's complement), AND, and OR operations. These operations will be described in greater detail later in this chapter.

C contains operators that directly correspond to the operations mentioned in the last three categories. In addition, C has many higher level operators derived from these last three groups. Also, there are several operators that deal directly with memory allocation and retrieval. C has no high level "operators" concerned with I/O, but it does have the **register** data class (chapter 7) that "encourages" special memory to CPU communication.

One of the primary reasons C is such a relatively flexible and powerful language is its inclusion of such a wide range of basic operators. The operators range from those that operate

---

* Strictly speaking, most computers are actually only capable of performing addition, complementation, rotation, and shift operations. The other basic arithmetic operations are derived from these.

on separate bits of a data type to those associated with one or more expressions or statements. The former variety of operator would naturally find greater usage in systems programming, while the latter variety often is employed in both high and low level programming.

A NOTE TO THE EXPERIENCED PROGRAMMER

The vast majority of C's operators may, when applied properly, be used with all of the fundamental data types. Many of these may be used on pointers as well (chapter 9). All of C's operators may be used on separate elements of an array and separate members of structures, provided that said element or member is of an acceptable data type (i.e. usually an optionally — extended fundamental or pointer data type). There are very few operators that can deal with these aggregate structures. Correct operand type is outlined in the "NOTES" section of each operand group.

A NOTE TO BEGINNERS

The C language has a reputation for brevity of syntax. This assertion is especially typified by its operators, as evidenced by table 4.1. All of the operators, except for **sizeof**, are defined by non-alphanumeric printing symbols. Instead of the command words BEGIN and END, as used in ALGOL, C makes use of the symbols { and } respectively. Similarly C's > symbol is equivalent to Fortran's .GTR. operator. This style economizes both typing time and source code length. Additionally, compile time is decreased because this use of fewer and different symbols leads to more efficient *parsing* by the computer. Parsing refers to the operation of recognizing and separating the different types of code (tokens) in a section of the program.

C's concise style may seem a disadvantage to the beginner or the occasional programmer, for at times sections of code may appear quite baffling. Fortunately with continued exposure, people usually become quickly accustomed to C's syntax. Therefore, one of the most important goals of this chapter in regards to the beginner is not the presentation of the specifics of every operator, but rather the presentation of the form and general usage of C's operators.

Although the application of some of the operators is more restricted or the details more complicated or involved, no attempt has been made to single out whole sections on these operators as advanced. At the least, the reader should familiarize himself or herself with the following operators or classes of operators, for they will be used extensively throughout the rest of this book:

- the simple assignment operator: =
- logical operators: || &&   !
- equality operators: ==   !=
- relational operators: <   >   <=   >=
- additive operators: +   -
- multiplicative operators: *   /   %
- the unary minus operator: -
- the decrement operator: --
- the increment operator: ++

The addition +, subtraction -, multiplication *, division /, simple assignment =, and parentheses () operators are used prematurely in some examples. These operators, with the exception of the assignment operator, =, correspond almost exactly to their algebraic counterpart. The reader is encouraged to use this chapter as a reference guide as questions on operator usage arise.

Table 4.1. Complete table of C's operators listed in order of precedence

Class Name	Operators	Associativity*
highest ↑ primary	() [] ->	→
unary	! ++ -- (cast) * & sizeof −	←
multiplicative	* / %	→
additive	+ -	→
shift	<< >>	→
relational	< > <= >=	→
equality	== !=	→
bitwise and	&	→
bitwise xor	∧	→
bitwise or	|	→
logical and	&&	→
logical or	||	→
conditional	?:	←
assignment	= += -= etc	←
lowest ↓ sequence	,	→

(Left margin spanning label: "highest → ... Precedence ... → lowest")

General Comments on Operators

CLASSIFICATION

Operators can be grouped either according to the number of operands they require or by the type of operation they perform. Utilizing the former scheme, C's operators may be divided into four categories:

- *Primary operators* — These define the punctuation and components of certain derived data types.

---

* → symbolizes left to right associativity while ← symbolizes right to left. (The binary operators *, +, &, |, and ∧ have undefined associativity.)

- *Unary operators*   These perform a specific function on only one operand.

- *Binary operators*   These require two operands. The majority of C's operators fall in this category.

- *Ternary operators*   This type of operator requires three operands. C's only ternary operator is the conditional operator, **?:** .

Although primary operators are associated with only one operand just as unary operators are, they differ from the latter in that they actually define an identifier. Specifically, primary operators single out elements of arrays (chapter 9) and members of structures (chapter 10).

The second classification scheme groups operators according to the type of operation they perform. In table 4.1, the binary operators are further divided using this scheme. (All of the operators from the multiplicative to the sequence operator, except for the conditional operator, are binary operators.) The ternary conditional operator is also obviously grouped under a class name reflecting its function.

PRECEDENCE AND ASSOCIATIVITY

All C operators have two attributes. An understanding of both of these is a prerequisite for their proper use. These two attributes are known as *precedence* and *associativity*.

Precedence refers to the rank or urgency of an operator. Most of us are familiar with the precedence of the common arithmetic operators. For example, the equation: x=6+4÷2 would simplify to x=8, because the division operation has precedence over the addition and equality operations.

Associativity refers to the order (direction) in which operators possessing the same precedence are to be executed. Again, utilizing an algebraic example, the equation y=6÷3*5÷2 would simplify to y=5 because the multiplication and division operators have equal precedence and are associated left to right. An equivalent equation would be y=((6÷3)*5)÷2 . If the associativity of these operators had been from right to left, the original equation would have simplified to y=.8.*

COMMUTATIVITY

All operators requiring more than one operand also possess either a commutative or a non-commutative flavor. An operator is said to be non-commutative if switching any two of its operands would change the value of the resultant expression. For instance, since 5÷3 does not equal 3÷5, the division operators would be of non-commutative flavor. With commutative operators, a change in the position of the operands will not affect the value of the resultant expression. The following C operators are commutative:

```
    *    +    ==    !=    &    ^    |
```

---

* Notice that reversing the associativity does not reverse the *commutative* structure of the equation. Therefore, if a right to left associativity was assumed incorrectly, the first task performed would be the division of five by two, 5÷2, and not the inverse, 2÷5.

ORDER OF EVALUATION [ADVANCED]

Normally the rules of higher precedence and direction of associativity will define the order of operations in an expression. However, two situations exist where the order of evaluation is undefined. In these instances, the compiler will dictate the specific order of evaluation.

The first instance involves the multiple use of one of a number of the following commutative binary operators:

$$* \quad + \quad \& \quad | \quad \wedge$$

According to the rules of associativity, in an expression involving several operators possessing the same precedence (multiple use of the same operator is a common case of this), the operations will be performed following associative rules. Unfortunately, many compilers will rearrange the operands in expressions involving the aforementioned operators so as to produce an order it considers the most efficient. In other words, while these five operands will follow rules of precedence, no such assurance exists concerning the observance of their right-to-left associativity.* The following simple expression:

X=5*3*9;

could be arranged by the compiler to give the equivalent statement:

x=9*5*3;

Some compilers will even ignore parentheses when performing this type of rearrangement. For those occasions where this kind

---

* One can alternately view the operators as possessing true right-to-left associativity, with the compiler rearranging the operands in an arbitrary order. The difference between these viewpoints is mostly one of semantics.

of liberal interpretation is undesirable, an extra step may be used to partially alleviate this problem.

```
a=5*3;
x=a*8;
```

Here, the first statement forces the multiplication of 5 by 3 to be performed first. Because variable **a** is used to store intermediate results, it is termed an *"intermediate temporary"*. Note that within each of the previous two statements, ambiguity still exists as to the order of evaluation. In the first statement, it is not known whether **5** is multiplied by **3** or vice-versa.

A similar condition exists for the evaluation of *subexpressions* and function calls. A subexpression is any proper string of operands and associated operators enclosed by a matched pair of parentheses. Consider the following statement:

```
x = (a - 9) / ( squ(c) + sin(c) ) ;
```

This statement contains two subexpressions, the second of which contains two function calls. System dependent criteria determine which of these subexpressions will be evaluated first. Similar criteria determine which function call within the second subexpression is given precedence. Refer to figure 4.1 for clarification. Each diamond shape denotes a system dependent decision, the arrows represent flow of direction, while the circle is a collection node. This figure portrays a rather unorthodox use of flowchart symbols. These symbols are generally used to develop program algorithms (see appendix E).
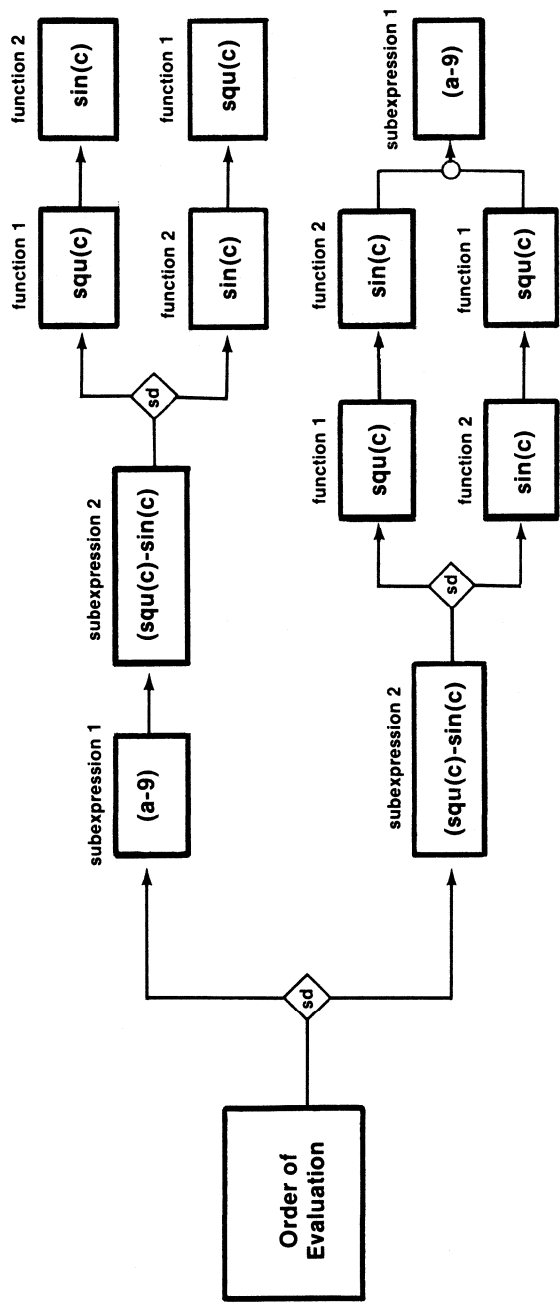
Figure 4.1. Order of evaluation flow of the equation
x = (a - 9) / (squ(c) - sin(c));

* All four possible evaluation flow lines are shown. All decision steps are system dependent.

The reader should not confuse the concepts of associativity and precedence with that of order of operand evaluation. In the following statement,

y=(exp 1) * (exp 2) + (exp 3);

the multiplication operator, *, has a higher precedence than the addition operator. Therefore, the multiplication operation will be performed before the addition. However, before these operations can be performed, the subexpression operands must be evaluated. As previously seen, this order is system dependent.

Why should the order of operand evaluation be of concern to a programmer? Normally it is not of concern because the same value will be returned regardless of the evaluation order. However, it is possible to code a statement where one expression (or function call) will change a value in another expression (or function call) and will therefore generally change the value of the evaluated expression (or returned value of a function). For instance, the following statements:

```
int y, x = 3 ;
y = (x * 3) + ((x=4) / 2) ;
```

would assign y the integral value 11 or 14. 11 would be assigned if the leftmost subexpression was evaluated first.

A value of 14 would result if the right hand subexpression involving the nested assignment expression was evaluated first. Function calls, nested assignment statements, and decrement and increment operators are capable of creating such system dependent duplicities. These duplicities are often referred to as "side effects".

Generally the modular, insulated structure of C precludes this type of ambiguous occurrence. However, in some instances such as that described previously, it may be necessary to use intermediate steps or temporary explicits. A closely related

problem occurs with argument list passing that will be discussed in chapter 6.

There are certain operators whose usage imposes a fixed order of operand evaluation. Four of these "forced evaluation order operators" follow:

&& || ,   **?:**

The first three of these operators have a forced left to right operand evaluation. The fourth operator, **?:**, has a conditional left to right evaluation. In the following sequence,

(exp 1) && (exp 2)

(sub) expression one will always be evaluated first, independent of the system.

The Reference Section

Most of the remainder of this chapter will consist of sections dedicately solely to the presentation of C's operators. These sections are written in a manner similar to that found in a reference guide. The operators are organized in what the author considered the most logical fashion — starting with operators dealing with bit manipulation (bitwise logical and shift operators); then to those that treat their operands as a single logical value (logical operators); to the "comparison" operators (relational and equality operators); then through the familiar arithmetic (additive and multiplicative operators) and assignment operators; progressing through the remaining unary operators and the ternary operator; and finally finishing with a discussion of primary operators which deal with data organization and group manipulation. Thus, the operators are generally organized from simplest to the more complex, from the computer's point of view.

The beginning of each section contains the operator's identification and syntax presentation, as well as a NOTES section. This latter section contains miscellaneous remarks and qualifications which are often explained in greater detail in the subsequent tutorial section. In the syntax section, parentheses are often subscripted with an "opt". This denotes that the inclusion of parentheses *might* be optional, depending on the precedence and associativity of the operators involved. The word *expression* denotes a number of resolvable tasks which might include variables, constants, and/or functions calls connected when necessary by operators. (A more definitive description of what constitutes an expression will be included in the section entitled "Primary Expression".)

The Binary Operators

The Bitwise Operators

Operator	Description	Syntax
~	bitwise NOT	~ (opt*expression*) opt
|	bitwise OR	(opt*expression*) opt | (opt *expression*) opt
&	bitwise AND	(opt *expression*) opt & (opt *expression*) opt
^	bitwise XOR	(opt *expression*) opt ^ (opt *expression*) opt

opt — earmarks optional characters.

NOTES

- The bitwise NOT operator, ~, has a considerably higher precedence than its peers.
- The bitwise operators accept only expressions resolvable to integer flavored (integral) data types.
- The bitwise operation yields an integral value of the same data type as its widest operand.

Bitwise (logical) operators manipulate the value of their operand(s) on a bit by bit basis, treating each bit position as a separate entity. The three binary bitwise operators (|, &, and ^) compare the corresponding bits of their operands, while the unary bitwise operator, ~, functions on one string of binary digits. All the bitwise operators require operands of integer flavor and all directly correspond to basic operators in the field of mathematics known as *boolean logic.*

~	NOT	(logical complement)
		OR	(logical disjunction or inclusive or)
&	AND	(logical conjunction)
^	XOR	(logical exclusive or)

Before these bitwise operators can be used effectively, the reader must possess a firm understanding of the underlying principles of boolean operations.

BOOLEAN ALGEBRA

The bitwise operators work by principles that are set forth in a branch of mathematics known as Boolean Logic. Boolean theory rests on the assumption that the conditions for certain logical problems may be stated in terms of one or more simple true or false statements related by the unambiguous logical operators *NOT, OR, AND,* and *XOR.* The binary digits 0 and 1 are used to define the logical conditions false and true respectively. For every possible combination of operator and condition(s), there exists a defined, constant outcome.

The logical NOT operator is the only unary logical operator. The NOT operation, also known as the logical complement, changes the truth value of its operand. For the condition M, NOT evaluates as:

NOT M: If M is true, then NOT M would be evaluated as false. Likewise, if M is false, then NOT M evaluates as true.

The OR operator (also called the "logical disjunction" or "inclusive or") functions as might be expected by its common use in English. The OR operation is defined for the two conditions M and N as:

M OR N: If M is true or N is true or both M and N are true, then M OR N equals true.

The AND (logical conjunction) operator also carries a rather intuitive meaning.

M AND N: If M is true and N is true, then M AND N equals true. Otherwise, M AND N equals false.

The XOR (exclusive or) operator is probably the most difficult operator to comprehend. It may help the reader to visualize the XOR operator as a special OR operator that excludes the true AND case.

M XOR N: If only one of the conditions M or N is true, then M XOR N equals true, otherwise M XOR N equals false.

Logical operators are commonly defined through *truth tables*. Truth tables list all condition combinations and outcomes possible for a given operator. Table 4.2 exhibits the Truth Tables for the NOT, OR, AND, and XOR operators. Note that these operations are commutative in nature.

Table 4.2. Truth Tables for the NOT, OR, AND, and XOR operators

Logical Complement

Condition	Outcome
M	**NOT M**
True	False
False	True

Logical Disjunction

Conditions		Outcome
M	**N**	**M OR N**
True	True	True
False	True	True
True	False	True
False	False	False

Logical Conjunction

Conditions		Outcome
M	**N**	**M AND N**
True	True	True
False	True	False
True	False	False
False	False	False

Exclusive Or

Conditions		Outcome
M	**N**	**M XOR N**
True	True	False
False	True	True
True	False	True
False	False	False

De MORGAN'S THEOREM AND IMP, EQV [ADVANCED]

DeMorgan's theorem states that the NOT operator and any other boolean operator may be combined to produce any desired outcome from a fixed set of known inputs.

This theorem is often written in one of the following algebraic forms:

```
NOT(M AND N) = (NOT M) OR (NOT N)
NOT(M OR N) = (NOT M) AND (NOT N)
M XOR N = (M OR N) AND (NOT (M AND N))
```

Therefore, a language need only include a minimum of two boolean operators to be complete.

Some languages also include the EQV (logical equivalence) and IMP (logical implication) operators. Inasmuch as C does not include these operators, the following equivalent operations may be substituted:

```
M EQV N = NOT (M XOR N)
M IMP N = (NOT M) OR N
```

Note that IMP is not commutative.

THE LOGICAL BITWISE MANIPULATION OF DATA

A value of false is represented in a computer by a value of zero, while a value of true is represented by any non-zero value. Since the discussion has centered on a unit bit, the value of true must be binary one as a binary bit must either be one or zero. The conditions previously discussed have been single true or false values, or pairs of values. The binary equivalent of this situation would be one bit for the NOT operation and a pair of bits for the AND and OR operations. But because the operand(s) of bitwise operators represent one or more bytes of data, a bitwise operation must be performed for every binary digit

allocated to the operand data type. Presented next are some examples of bitwise operations on certain integer values. An 8-bit integer storage length and two's complement format has been assumed.

EXAMPLE 1: W = ~26;

0	1	2	3	4	5	6	7	bit number

0	0	0	1	1	0	1	0	Operand = 26_{10}

1	1	1	0	0	1	0	1	W = NOT 26_{10} = 229_{10} or -27_{10}

This is an example of a bitwise NOT operation. Since all one values (true) were changed to zeroes (false) and vice versa, this operation could also be termed *one's complement*. If the result is interpreted as an unsigned integer, then it will have the value 229_{10}. The subscript 10 denotes the base.

If the leading bit is interpreted as the sign extension, then the value will be interpreted as the *two's complement* form of the negative number -27. This assumes, of course, that the computer system employs a two's complement pure binary format. Irrespective of the numeric interpretation, the bit pattern remains consistent.

EXAMPLE 2: X = 26 | 9;

0	1	2	3	4	5	6	7

0	0	0	1	1	0	1	0	Operand a = 26_{10}

0	0	0	0	1	0	0	1	Operand b = 9_{10}

0	0	0	1	1	0	1	1	x = 26_{10} OR 9_{10} = 27_{10}

EXAMPLE 3: y=26 & 9;

0	1	2	3	4	5	6	7
0	0	0	1	1	0	1	0

Operand a = 26_{10}

0	0	0	0	1	0	0	1

Operand b = 9_{10}

0	0	0	0	1	0	0	0

y = 26_{10} AND 9_{10} = 8_{10}

EXAMPLE 4: z = 26 ^ 9;

0	1	2	3	4	5	6	7
0	0	0	1	1	0	1	0

Operand a = 26_{10}

0	0	0	0	1	0	0	1

Operand b = 9_{10}

0	0	0	1	0	0	1	1

z = 26_{10} XOR 9_{10} = 19_{10}
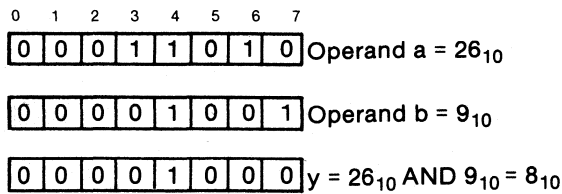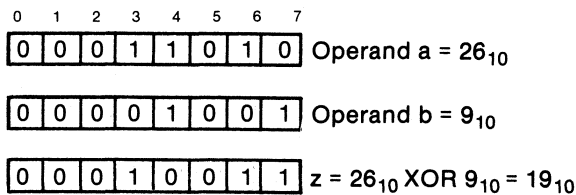
Examples 2 through 4 demonstrate the OR, AND, and XOR operations respectively. Once the values have been converted to binary format, the operations become quite obvious. Negative values could again complicate matters somewhat. Therefore, sign suppression via the **unsigned** extension is usually preferable.

Shift Operators

Operator	Description	Syntax
<<	(bitwise) left-shift	(opt *expression 1*) opt << (opt *expression 2*) opt
>>	(bitwise) right-shift	(opt *expression 1*) opt >> (opt *expression 2*) opt

NOTES:

- The specifier (expression 2) must evaluate to a value having the following specific properties:

  1. It must be of integral data type flavor.
  2. It may not be a negative value.
  3. It may not be equal to or greater than the data type length in bits of the returned value of *exp1*.

- The expression *exp1* must evaluate to an integer data type flavored (integral) value.

- Results of a right-shift operation on a signed quantity are system dependent.

- The data type of the result will be that of the left operand (object).

The shift operators $<<$ and $>>$ perform what may seem to an applications programmer an unusual function. Shift operators actually shift the bits in the integer flavored data object over the amount determined by the specifier. Vacated bits in positive object operands are filled with zeroes. For negative object values, these vacant bits are, in effect, filled with ones on a machine using a complement format. An eight bit storage length and two's complement binary representation has been assumed for the sake of convenience. A subscript of ten denotes a decimal value.

EXAMPLE 1: r = 23 $>>$ 2;

0	1	2	3	4	5	6	7	bit number

0	0	0	1	0	1	1	1	Object=23_{10}

0	0	0	0	0	1	0	1	$r=23_{10} >> 2_{10} = 5_{10}$
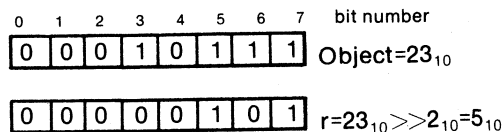
In this example the object value is right-shifted two bits or positions and the vacated bits are filled with zeroes. The need for binary representation of data values should be obvious to the reader.

EXAMPLE 2: s = $5 << (1 + 2)$;

```
 0   1   2   3   4   5   6   7
[0 | 0 | 0 | 0 | 0 | 1 | 0 | 1]  Object = $5_{10}$

[0 | 0 | 1 | 0 | 1 | 0 | 0 | 0]  s = $5_{10} << 3_{10} = 40_{10}$
```

EXAMPLE 3: s = $42 << 2 + 1$;

```
 0   1   2   3   4   5   6   7
[0 | 0 | 1 | 0 | 1 | 0 | 1 | 0]  Object = $42_{10}$

[0 | 1 | 0 | 1 | 0 | 0 | 0 | 0]  s = $42_{10} << 3_{10} = 80_{10}$
```

Examples 2 and 3 demonstrate the use of the left-side operator. In the second example, no parentheses were used to force associativity of the addition operands. Here their inclusion is optional because the addition operator has a higher precedence than the shift operator. Also, note that a binary one has been shifted "off the left edge" in example 3 and has been permanently lost. Subsequent right-shifting of variable **s** will not recover this digit.

EXAMPLE 4: t = $86 << 1$;

```
 0   1   2   3   4   5   6   7
[0 | 1 | 0 | 0 | 1 | 1 | 1 | 0] Object = $86_{10}$

[1 | 0 | 0 | 1 | 1 | 1 | 0 | 0] t = $86_{10} << 1 = 156_{10}$ or $-200_{10}$
```

In example 4, the object value is simply left-shifted one place. A complication arises in that the leftmost (highest order) bit is reserved for a sign extension in complement formats. The unsigned interpretation of this binary string is decimal 156. Alternately, this string represents -100 in two's complement format and -99 in one's complement (see appendix C).

For negative quantities, the vacated bit positions in complement formats are filled with ones.

EXAMPLE 5: v = -50 $<<$ 2;

0	1	2	3	4	5	6	7	
1	1	0	0	1	1	1	1	Object = -50_{10}

0	0	1	1	1	1	1	1	v = $-50_{10} << 2 = 73_{10}$
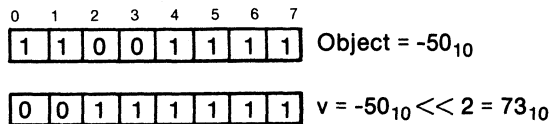
Bit position seven is the sign extension bit. In this example, a binary one in this position signifies a negative number in two's complement format. When the two's complement form of decimal -50 is left-shifted two positions, the sign extension will be filled with a zero. This indicates a positive number in simple binary format. The vacated positions to the right have been filled with ones because the value was originally negative. The result of this series of steps is the positive integer value 73. Note how the highest order binary zero has been "sacrificed" to change the sign extension.

It is important to remember that these examples (as well as several future ones) assume an eight bit integer memory storage length. This assumption is normally incorrect. In fact, the sign extension bit is usually located on or past the fifteenth bit position on the majority of computers.

Logical Operators

Operator	Description	Syntax
!	logical negation	! (opt *expression*) opt
&&	logical AND	(opt *expression*) opt && (opt *expression*) opt
‖	logical OR	(opt *expression*) opt ‖ (opt *expression*) opt

NOTES

- The unary logical negation operator, !, possesses a much higher precedence than the other logical operators.
- The logical AND and OR operators, && and ‖, have a forced left-to-right order of evaluation.
- In C, the logical AND and OR operators possess an attribute known as "short-circuit evaluation".
- A true outcome yields a value of int 1, while an outcome of false yields a value of int 1.
- The logical operators accept operand expressions resolvable to an [extended] fundamental data type or pointer value.

Logical operators function in much the same fashion as the corresponding logical bitwise operators except that the logical operators treat their operand(s) as single units.

Here, a false condition is again described by a zero value, but a true condition is described by any non-zero value. The outcome (i.e. the value returned by the entire logical operation) is a zero int value if the operation evaluates as false. int one is associated with a true outcome. The operands must be expressions resolvable to a pointer value or to a constant of fundamental data type flavor (optionally extended). The examples presented in this section will again only be one byte long for the purpose of convenience.

LOGICAL NEGATION

The logical negation operation resolves to a true value (one) if the operand is false (zero), while the operation evaluates to zero if the operand is any non-zero value. This is shown in the following example:

w = !26;

0	1	2	3	4	5	6	7	bit number
0	0	0	1	1	0	1	0	Operand = 26_{10}

0	0	0	0	0	0	0	0	w = NOT 26_{10} = 0
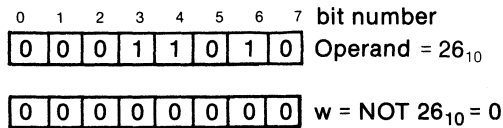
Because the operand was a non-zero value, the operation resulted in assigning variable **w** a zero value. Compare this example with the example presented using the bitwise NOT operator. There is no ambiguity associated with the use of negative numbers as all non-zero operand values are treated identically as true values.

LOGICAL OR

The logical OR operator returns a value of true (int one) when one or both of its operands evaluates to true (non-zero), and a value of false (int zero) when both of its operands evaluate to false (zero). The logical OR operation will rarely return a value equivalent to its bitwise counterpart. This will in fact only occur when one operand has a value of int1, and the other has a value of int 1 or 0.

$y = 26 \;\|\; 0 ;$

	0	1	2	3	4	5	6	7	
	0	0	0	1	1	0	1	0	Operand a = 26_{10}

0	0	0	0	0	0	0	0	Operand b = 0

0	0	0	0	0	0	0	1	y = $26_{10} \| 0 = 1$

In the preceding example, operand **a** is non-zero (a condition of true). Therefore, the value of the outcome will be true (int one) regardless of the second operand's value.

LOGICAL AND

The logical AND operation returns a value of true (int one) if both of its operands evaluate to true (non-zero). A value of false (int zero) is returned if one or both of its operands evaluate to false (zero).

EXAMPLE 1: $x = 26 \;\&\&\; " ;$

	0	1	2	3	4	5	6	7	
	0	0	0	1	1	0	1	0	Operand a = 26_{10}

0	0	0	0	0	0	0	0	Operand b = 0 = ASCII *null*

0	0	0	0	0	0	0	0	x = $26_{10} \&\& 0 = 0$

EXAMPLE 2: $x = 26 \;\&\&\; 9 ;$

	0	1	2	3	4	5	6	7	
	0	0	0	1	1	0	1	0	Operand a = 26_{10}

0	0	0	0	1	0	0	1	Operand b = 9_{10}

0	0	0	0	0	0	0	1	x = $26_{10} \&\& 5_{10} = 1$

Notice that in example 1, the logical AND operates exactly the same as its bitwise counterpart. This observation breaks down in the second example as the returned value is one. Here, a bitwise AND would have returned a value of eight.

EVALUATION OF OPERANDS [ADVANCED]

The logical AND and OR possess two attributes that can be utilized to considerable advantage by the programmer concerned with minimizing run time. The first attribute, referred to as "force order of evaluation", ensures that the associativity governs the order of evaluation. In other words, for these two logical operators, the expression on the *left* is always evaluated before the expression on the right. The second attribute, known as "short-circuit evaluation", promises that expression evaluation stops as soon as the truth value of the logical expression is known. Thus, a left operand with a true value will short-circuit an | | operation, resulting in a value of true (one). Similary a left operand evaluation of false will short-circuit an && operation, resulting in a false (zero) returned value.

Relational and Equality Operators

Operator	Description
$<$	less than
$>$	greater than
$<=$	less than or equal to
$>=$	greater than or equal to
$==$	equality (equal to)
$!=$	inequality (not equal to

Relational   Equality

($_{opt}$ *expression 1*)$_{opt}$ *OPTR*
  ($_{opt}$ *expression 2*)$_{opt}$

where *OPTR* represents one
of the listed operators

NOTES

- The equality operators have a precedence immediately below that of the relational operators.
- Expressions 1 and 2 must evaluate to an [extended] fundamental data type or pointer value.
- A true outcome is represented by a value of int one while a false value is represented by int zero.
- The relational and equality operators are sometimes collectively referred to as the comparison operators.

Relational and equality operator usage is generally associated with control flow statements, although they can be utilized in other situations. Both the relational and equality operators are used to compare the magnitude of two values. If the comparison is true, an integer value of one will be returned. If the comparison is false, then the integer value of zero is returned. While the comparison operators are usually associated with the ternary conditional expression or control flow statements, they can be validly used in any expression.

$$e = 5 + (a == 2);$$

The variable **e** will be assigned the value decimal 6 if the value of variable **a** is exactly two, otherwise **e** will be assigned the value 5. This is a direct result of the fact that the subexpression **(a == 2)** (i.e. "**a** equals 2") must evaluate to either true (int 1) or false (int 0).

The preceding example illustrates two interesting points. The reader should differentiate between the equality operator == and the assignment operator =. The latter operator is used to assign a value to a variable. The equality operator is used to compare two values, and evaluates to a value of one or zero. The equality operator does not change the value of any variable. The other point concerns the use of the pair of parentheses in the equation. Without these parentheses, the variable **a**

would associate with the additive operator **+**, resulting in a statement equivalent to:

$$e = (5 + a) == 2 ;$$

e would now either be assigned the value of 1 or 0 depending on whether the subexpression **5 + a** was or was not equal to **2**, respectively.

Other than the problems with precedence and confusion with other operators, use of the comparison operators is quite straightforward. One need only substitute the operator description in its place and form a question.

$$(f >= g)$$

The expression is readily interpreted as: "Is variable **f** greater than or equal to variable **g**?" An answer of "yes" or "true" translates to a value of integer one while that of "no" or "false" is represented by integer zero.

C allows the use of constant operands as well as variable terms in either or both sides of relational and equality expressions. The latter possibility, exemplified by an expression such as **3 < 5** is never used, since the truth value of the expression never changes (i.e. a value of 1 may therefore be substituted for this expression).

Additive Operators

Operator	Description	Snytax
+	addition	$(_{opt}$ *expression* $)_{opt}$ + $(_{opt}$ *expression* $)_{opt}$
-	subtraction	$(_{opt}$ *expression1* $)_{opt}$ - $(_{opt}$ *expression2* $)_{opt}$

NOTES

- The addition operator has an undefined associativity, as operands or expressions involving a string of additions may be arbitrarily arranged by the compiler.

- An additive operand can be any expression that resolves to an [extended] fundamental data type or pointer value.
- Pointer arithmetic is concerned with the manipulation of memory addresses utilizing unit data object lengths (chapter 9).

The reader should already be familiar with these two operands from basic math. The additive operators have an equal precedence below that of the multiplicative operators and the unary minus. The subtraction operation should not be confused with the unary minus operation, despite the fact that both of these are represented with the same symbol. Data type promotion occurs, of course, whenever the two operands are not of identical type.

Multiplicative Operators

Operator	Description	Syntax
*	multiplication	$(_{opt}$ *expression* $)_{opt}$ * $(_{opt}$ *expression* $)_{opt}$
/	division	$(_{opt}$ *expression 1* $)_{opt}$ / $(_{opt}$ *expression 2* $)_{opt}$
%	modulus (division)	$(_{opt}$ *expression 1* $)_{opt}$ % $(_{opt}$ *expression 2* $)_{opt}$

NOTES

- The multiplication operator, *, has an undefined associativity, or alternately has a set left to right associativity, while the compiler is free to rearrange operands.
- For the * and / operations, the operands can be any expression yielding an [extended] fundamental data type value.
- The modulus operation requires operands of integer type flavor (integral).
- The unary minus has higher precedence than these equal precedence multiplicative operators.

Because the multiplication operator, *, functions as one would expect, nothing more will be said about it. The division operator will also function normally, except in situations where both operands are of integer type flavor. In such an instance, any fractional remainder will be truncated. Two examples are shown on the next page:

9/5 *evaluates to* 1

9.0/5 *evaluates to* 1.8

-9/5 *evaluates to* -1

The modulus operator, %, yields the remainder from the division of two integral values. We can express this operation as:

a%b ⇨ a-(a/b)*b

where **a** and **b** are integral values and **b** is not zero. Therefore:

9%5 *evaluates to* 4

-9%5 *evaluates to* -4

9%-5 *evaluates to* 4

9%3 *evaluates to* 0

Unlike normal division, modulus division (sometimes called modulo arithmetic) yields a value whose sign is either always the same as the dividend (expression 1) or a value of zero. The multiplicative and additive operators are often jointly referred to as the arithmetic operators.

Assignment Operators

Operator	Description	Syntax
=	(simple) assignment	*lvalue* =($_{opt}$*expression*)$_{opt}$
*=, /=, %=, +=, -=, &=, ^=,|=, >>=,<<=	compound assignment	*lvalue OPTR* ($_{opt}$*expression*)$_{opt}$

where OPTR is one of the listed compound assignment operators

NOTES

- When necessary, the data type of the expression on the right side will be demoted or promoted to conform to the data type of the lvalue. (The details of this process are outlined in chapter 3's "Type Conversion" section.)
- All assignment operators accept [extended] fundamental data type values for their right side operands.
- The left side operands must be lvalues. Lvalues are symbols that reference a string of memory (i.e. an object). All assignment operators accept the following lvalues:

    1. fundamental data type variable identifier
    2. extended fundamental data type variable identifier
    3. array and subscript identifier (array element)
    4. structure variable and member identifier (structure element)
    5. an expression resolvable to one of the above objects

    In addition, the =, +=, and -= operators allow pointer and pointer-derived data types as lvalues.

Although a more thorough discussion of lvalues will be presented in chapter 9, a brief description is in order. Memory must be addressed by a programmer by either explicitly or implicitly describing a memory location. High level programmers do this implicitly using symbolic names (variable identifiers). The computer then associates these identifiers or names with specific blocks of memory it uses to store values assigned to said variables. Any token that denotes such a memory location (and the actual value of the address) is known as an lvalue (pronounced "ellvalue"). Pointers, [extended] fundamental variables, array elements and structure members are all lvalues. The term lvalue originates from the assignment

operation. Any token capable of being placed on the *left* side of the assignment operation must reference a memory block in storage; it is therefore an lvalue. C, like all other high level languages, does not allow the applications programmer to specify exact numeric addresses to be assigned to variables.

With this implicit addressing of memory, the variable-as-a-box concept is, in fact, a fairly realistic model. In this concept, a block of memory referenced by a variable name is thought of as a box. When a value is assigned to a variable, that value is altered to fit (type converted) the block of memory referenced by the variable identifier. This altered value is then stored at this location. Any previous information held at this location will be lost.

The simple assignment operator has been used several times already. The reader should be aware of the difference between the assignment operator = and the equality operator ==. The former is used to transfer information to a variable (region of memory). The latter compares two values, returning the outcome of said comparison.

The ten other compound assignment operators all combine a binary operation with a simple assignment operation. Therefore, the expression:

lvalue OPTR = (expression)

is equivalent to the expression:

lvalue = lvalue OPTR (expression)

More concretely if the variable **b** was initialized to a value of five, then each of the following statements would change **b**'s value to thirty:

```
    b *= 4 + 2;        /*equivalent*/
or  b= b * (4 + 2);   /*statements*/
```

Note that parentheses are not required in the first expression even though the implied multiplication normally would have precedence over the addition. The compiler automatically "supplies" these parentheses —explicit parentheses around the whole expression are entirely superfluous.

Compound assignment operators save coding time, are generally easier to read, and can possibly produce more efficient object code.

The Comma Operator

Operator	Description	Syntax
,	comma	*expression,expression*

NOTES

- The comma operator has a left to right forced order of evaluation.

The comma operator serves a function somewhat related to the semi-colon in that it separates code. However while the semi-colon is considered punctuation, the comma is normally an operator. When a pair of expessions is separated by a comma, the following occurs:

1. The left expression is evaluated.
2. The result of the evaluation of the left expression is discarded.
3. The right expression is evaluated and this result is the value and the data type of the entire comma expression.

For example, the statement:

sum = (3.75, I % 9, 300);

assigns the value of integer 300 to variable **sum**. The first two subexpressions (ie. **3.75** and **i % 9**) are evaluated and discarded. Obviously here the comma operations have served no purpose except to waste computation time. Normally, the subexpressions contain assignments so that when the result is discarded, it is still retained by the specified value. For example, the following expression assigns three unrelated values to three different variables:

sum = 30,i%=9,tax=TAX

Because this multiple assignment is still only a single expression, it can be utilized wherever one expression is required. Furthermore, postfixing the expression with a semicolon will transform it into a valid statement.

Comma within argument lists should not be thought of as operators but rather punctuation because these commas do not ensure left to right evaluation. Thus the statements:

```
int i = 10;
printf ("|n %d, %d", i, ++i);
```

will result in one of the following two lines being output, depending on the compiler:

10, 11 *or* **11, 11**

To ensure against a "side effect" of this sort, intermediate temporary variable(s) should be used.

The Unary Operators

The Indirection and "Address Of" Operators

Operator	Description	Syntax
*	indirection	$*(_{opt}$ *expression* $)_{opt}$
&	address of	$\&(_{opt}$*lvalue*$)_{opt}$

NOTES

- The indirection operator, *, accepts only expressions resolvable to a pointer value. This value should be within the limits of usable memory.

- The "address of" operator accepts variable identifiers of [extended] fundamental data type and array elements. Register variables cannot be used (chapter 7).

- The indirection operator, *, may be used on either side of an assignment statement (e.g. *a = *b), while the "address of" operator, &, can only be utilized on the right side of this sort of statement (e.g. c = &d).

These two operators enable the use of *pointers* in C. A pointer is a variable that "points to" or contains the memory location of (usually) another variable. The address of operator, &, yields the memory location of the subsequent [extended] fundamental variable, structure member or array element. For example, in:

ptr1 = &var1;

The address of operator acts upon the variable **var1**, yielding the numeric value of the variable's starting memory location. This value is assigned to variable **ptr1**. As mentioned earlier, the memory location associated with a variable is termed the *lvalue*. When the computer references a variable, it uses the

variable's associated lvalue to locate the starting memory address of the section of memory reserved for that variable. The computer (CPU) then fetches the numeric content of this section of memory. This numeric content is what generally concerns the high level programmer. This is the value (termed the *rvalue*) held by this variable.

In the preceding example, the lvalue of **var1** will be assigned as the rvalue of **ptr1**. In other words, the variable pointer **ptr1** holds the starting address of variable **var1**.

An address is not of much use without a means of fetching information from the corresponding memory section. Here, the indirection operator is useful. This operator retrieves the contents of the memory address specified by the subsequent pointer operand. This pointer operand can be a literal value. However, since the programmer generally does not know the specific memory address of a variable beforehand, the indirection operator is generally used in conjunction with the address of operator. The following two lines are equivalent to the statement **var2 = var1;**:

```
ptr1 = &var1 ;
var2 = *ptr1 ;
```

The first line assigns the address of **var1** to **ptr1**. The second line assigns the contents at the memory address specified by the pointer **ptr1** (i.e. the lvalue of **var1**) to the variable **var2**. By this process **var2** is indirectly assigned the value of **var1**.

The advantage of pointers lies in their inherent global nature and more efficient access to elements of derived data types such as structures and arrays. A more complete description of pointers is presented in chapter 9.

The reader should keep in mind that different [extended] fundamental data types have different memory storage allocation requirements. Therefore when a pointer is used, the starting address of an [extended] fundamental variable is a neces-

sary but insufficient parameter. With just this information, neither the computer nor the programmer would be able to ascertain where the section of memory referenced by a pointer ended (i.e. "is it one, two, four, or more bytes long?"). Also, they could not ascertain what type of data it contained (i.e. int, long int, float, etc.).

To circumvent these problems, it is necessary to declare pointers before their use. Pointers are declared in much the same way as other data types, except that pointers are declared by the data type they point to rather than their actual data type. (C makes no specification as to the length of pointers, although they will usually be stored as int, unsigned int, or long int data type.) Suppose a pointer to an array of four integers, a pointer to an integer, and an integer variable were to be declared. This could be accomplished as follows:

int (*ptr_ary)[4], *ptr_int, ard;

Note that the indirection symbol is used here, not to denote the indirection operation, but to differentiate between pointers and the related non-pointer data type. For example without the * symbol, the variable **ptr_int** would be a simple int variable despite the implications of its name.

Although a discussion of arrays will be delayed until chapter 9, the reader should note the use of parentheses in the first declaration. If these were missing, the identifier would associate or "bond" more closely with the brackets because this operator pair has higher precedence than the indirection operator. Without the parentheses the first identifier would be interpreted as an array of four pointers to four integers. The second declaration defines **ptr_int** to be a (variable) pointer to an integer. Finally **ard** is declared a integer variable. The declaration of fundamental and associated derived data types in the same statement is allowed and commonly used in C.

The Unary Minus Operator

Operator	Description	Syntax
-	unary minus	- $($opt $expression$ $)$opt

NOTES

- The unary minus operator accepts expressions resolvable to an [extended] fundamental data type.
- The unary minus sign may not be used on the left side of an assignment expression (i.e. on an lvalue).

The unary minus gives the negative of its operand when the operand is a signed quantity. This operation usually involves simple sign reversal for floating point data types, while for signed integral data types (two's) complementation is typically performed. When applied to an unsigned quantity (i.e. fundamental data types using the unsigned extension), the operation yields the value: (2^n) - Operand, where n is the number of bits of storage allocated to the integral operand.

Assuming a two byte (n=16 bits) int storage allocation, the following section of code would assign a value of decimal 63,663 to the variable **b**:

```
int a = 1873 ;
unsigned int b = -a;
```

The assignment may be represented as follows:

Decimal	Binary
65,536 (2^{16})	1,0000,0000,0000,0000
-1,873	-111,0101,0001
63,663	1111,1000,1010,1111

Note that the binary equation operation procedure is equivalent to taking the two's complement of the subtrahend.

The unary minus operator is differentiated from the subtraction operation by its syntax. In the absence of a preceding expression or in the presence of a preceding expression and binary operator, the hyphen symbol will be treated as a unary minus. However the following expression:

a = b--c;

may elicit an error condition on some compilers because the subtraction/unary minus operator pair is confused with the decrement operator. Addition should be substituted in this case.

The Increment and Decrement Operators

Operator	Description	Syntax		
++	increment	++*lvalue*	or	*lvalue*++
--	decrement	--*lvalue*	or	*lvalue*--

NOTES

- These operators will accept the following lvalues:
  1. an identifier of [extended] fundamental data type
  2. an identifier referencing specific elements in arrays and structures (remember C has few operators that deal with composite data types)

These two relatively unusual operators are commonly employed in C code. The increment operation adds one to its operand, while the decrement operation subtracts one from the operand. When these operators are placed before their operand, they are termed *prefix* operators. Placement after the operand qualifies them as *postfix* operators. The positioning of the operator does not change the effect of the operator. Instead, it defines when this operation will be performed. The

subexpression **++a** increments **a** before further parent expression evaluation, while the subexpression **a++** increments variable **a** after all current line evaluations have been completed.

Contrast the following two sections of code:

```
/*section1*/              /*section2*/
int a = 4;                int a = 4;
b = ++a;                  b = a++;
```

The righthand section of code assigns a value of four to **b**, because the variable **a** is incremented only after the assignment operation has been carried out. Therefore, the statement:

```
b = a++;
```

is equivalent to either of the following two sections of code:

```
b = a;            b = a;
          or
a = a + 1;        a += 1;
```

On the other hand, variable **b** is assigned the value of five in code section 1. The increment operation occurs before the assignment, because it is used in postfix position and it has higher precedence than the assignment operator. The statement:

```
b = ++a;
```

is equivalent to either of the following:

```
a = a + 1;        a += 1;
          or
b = a ;           b = a;
```

Increment and decrement operators can only be applied to single variable lvalues. An expression such as **(a+b)++** would be illegal because there is no stipulation as to which of the variables is to be incremented. The whole expression **(a+b)** cannot be incremented, because it represents a value and not a variable

lvalue. The reader should remember that the increment and decrement operators, unlike most of the other operators in C, actually change the value of a variable due to the implied assignment.

The Bitwise NOT and Logical NOT Operators

The bitwise NOT operator (one's complement), $\sim$, is discussed in the section entitled "Bitwise Operators", while the logical NOT operator, ! is discussed in the section "Logical Operators".

The Sizeof and Cast Operators

Operator	Description	Syntax
(dtn)	cast	*(data type name)* ($_{opt}$ *expression*)$_{opt}$
sizeof	sizeof	**sizeof** ($_{opt}$ *expression*)$_{opt}$
	or	**sizeof(** *data type name* **)**

NOTES

- The sizeof and cast operators accept operand expressions resolvable to [extended] fundamental data type (or pointer) values. Also, the size of operator will accept array identifiers and parenthesized data type names.
- The sizeof operation returns an int value.
- The cast operation returns the value of its operand type converted to the data type specified.

The cast operator is a fairly useful operator, especially when applied to formal argument passing (chapter 6). This operator is used to force or coerce the type conversion of the operand to the [extended] fundamental data type specified within the operator. Type conversion rules were detailed towards the end of the last chapter, in the section titled "Type Conversions [Advanced]". The type conversion operation is

equivalent to that performed across an assignment statement, except that no actual assignment takes place. For example, the expression:

(int) .3597e02

yields the integer value 35, since the fractional part of a floating flavored value is truncated when demoted to an integer flavored data type. The actual rvalues of variables are not changed by this operation. The code section:

int a;
double b = 123.45678;
a = (int)b;

assigns the integer value 123 to the variable **a**. The value of **b** is unchanged in this section.

The **sizeof** operator also deals with data type memory allocation, although this operator is used principally by systems programmers. The **sizeof** operation yields the size, in bytes, of its operand. The "returned value" is of **int** data type. C does not stipulate the length of a byte, although a byte is commonly eight bits. If **int1** is a declared integer variable and if integers have a two byte memory allocation on the present system, then the following expressions would yield values of integer 2:

sizeof int1   *or*   **sizeof (int1)**

The **sizeof** operator can be used on uninitialized variables.

sizeof, unlike most other operators, can be applied to whole arrays. This operation yields the number of bytes used by an array. This operator may also be applied to array elements. It then yields the number of bytes allocated to each element in the array (every array element must be of the same data type).

When the operand is a pointer variable, the result is once again the integer number corresponding to storage length, in bytes, of the operand. Pointers will either be allocated two bytes or four bytes (corresponding to unsigned int, int, long int). However, when the operator is a pointer variable prefixed with the indirection operator, *, the number of bytes allocated to the object data type referenced by the pointer will be returned.

The operand may also be a constant, variable, or expression combining any of the previously mentioned data types. The program below illustrates the effect of the sizeof operator on different data types:

```
/*      Program 4.1 - Effect of sizeof
    operator on different data values */
main()

{   int a,b,c,ary[8],d,e,f,g,h;
    long NUM = 12345678;
    long *ptr;
    ptr = &NUM;
    a = sizeof ptr;
    b = sizeof *ptr;
    c = sizeof ary;
    d = sizeof ary[4];
    e = sizeof(.5673938467 + 1);
    f = sizeof .67e05;
    g = sizeof 'e';
    h = sizeof "Hello";
    printf("\n\na=%d, b=%d,\tc=%d, ",a,b,c);
    printf("d=%d,\ne=%d, f=%d, ",d,e,f);
    printf("\tg=%d, h=%d",g,h);
}


    a=2, b=4,        c=16, d=2,
    e=8, f=8,        g=2, h=6
```

Program 4.1 assigns the number of bytes of storage length of several variables and constants to variables **a** through **h** and then outputs these values. These single letter variables were pragmatically declared as int's, although these small integral values could be stored exactly in any of the integer flavored data types or approximately in any of the floating data types.

The result of a sizeof operation is machine dependent. The reference computer for this text conforms to the statistics listed in table 3.2. Variable **a** is assigned a value of 2 because pointers are int's which are allocated two bytes on the reference system. **b** is assigned a value of 4 because the symbolic constant **NUM**, which was indirectly referenced by the pointer **ptr** in line 9, is a long int. The next two variables, **c** and **d**, are assigned the "size of" the whole array **ary** and the fifth element of the array respectively. An array size is the size of one element times the number of elements.

Variables **e** and **f** are assigned the value 8 because the sizeof operands are both double constants. The parentheses in line 14 are necessary, because the sizeof operator has higher precedence than addition. The expression is resolved to double as outlined in the section entitled "Data Type Conversions" in this chapter. The constant operand of line 15 fits the format of the float data type, but the value is automatically promoted to double as noted in chapter 3.

Automatic char to int promotion explains the assignment of the value 2 to variable **e**. The string **Hello**'s size of six may be explained as follows. Each of the five characters has a length of one. These char values are not promoted to int's, because they are string (or array) members. The sixth member of this string is an invisible end marker (see chapter 9). Compilers may differ slightly on their criteria of automatic promotion, especially for array elements.

The second syntactic form of the sizeof operator allows the application of this operator to data type names. The opera-

tion yields the number of bytes that the named data type is allocated on the current system. The result is in int form. For example, the statement:

d_bytes = sizeof (double);

would assign, on this text's reference system, a value of 8 to **d_bytes**.

sizeof expressions are treated as constants by the compiler, and therefore may be used where constants are required: external variable initializers, conditional directive comparison expressions, case labels, etc. The expression **sizeof** (*exp*) is taken to be a unit by the compiler, so it is equivalent to **(sizeof (*exp*))**.

Ternary Operator

The Conditional Operator

Operator	Description	Syntax
?:	conditional	(opt *expression 1*) opt **?** (opt *expression 2*) opt **:** (opt *expression 3*) opt

NOTES

- Although a conditional expression (i.e. an expression formed by the concatenation of the conditional operator with three subexpressions) mimics the if-else conditional statement, it is a true expression and may be used as any other expression can.

- The conditional operator has a fixed order of evaluation. Specifically, expression 1 is evaluated first. Then if expression 1 resolved to true (non-zero), expression 2 is evaluated, otherwise expression 2 was false (zero) and expression 3 is evaluated.

- The result of the entire expression is type converted in accordance with subexpressions 2 and 3 (i.e. whichever has the wider data type regardless of which was evaluated).

The conditional operator is very unusual, in that it serves a function much like that served by the **if-else** control flow statement (chapter 5). The conditional expression may be interpreted as follows. Expression1 is evaluated first. If expression1 is true (non-zero), then expression2 will be evaluated. Expression 2's value will be the resultant value of the entire conditional expression. If expression1 is false (zero), then expression 3 will be evaluated. Expression 3's value will be the resultant value of the entire conditional expression. For example, the following statement assigns a value of 10 to the variable **y**:

$$y = 15 \ ? \ 10 : 13 \ ;$$

Since the literal **15** is true (non-zero), expression2 is "evaluated" and is the value yielded by the conditional expression. This value is then assigned to the variable **y**.

The above example is rather absurd, because the conditional expression will always evaluate to 10. This value is equivalent to and may be substituted for the conditional as follows:

$$y = 10 \ ;$$

Pragmatically, expression1 is always comprised of at least one unknown quantity or variable. Often these tokens are combined with relational and/or equality operators to form an expression. The following statement assigns the absolute value of variable **c** to variable **b**:

$$b = (c >= 0) \ ? \ c : -c \ ;$$

If **c** is a negative quantity, the subexpression **(c>=0)** returns a value of false, resulting in **y** being assigned the value of (sub)-expression3. Because the value of **c** was logically tested as negative, the value assigned to **b** will be positive (i.e. the unary minus of a negative quantity is always positive). If the variable **c** holds a positive value, expression1 resolves to true, and the value of **c**, which must be positive or zero, is assigned to the variable **b.** Parentheses were not actually needed around the relational expression1, as the conditional operator has very low precedence.

The conditional expression has a fixed order of evaluation. expression1 is always evaluated first. If it is true, then expression2 is evaluated while expression3 is not. If expression1 is false, then expression3 is evaluated while expression2 is not. In the following conditional expression, either **x** or **w** will be incremented, but not both, depending on the truth value of expression1:

expression 1 **? ++w : ++x**

Thus, the conditional operator is said to employ "short circuit evaluation".

Subexpressions 2 and 3 are also type converted as outlined in a later section of of this chapter entitled "Type Conversions". If **flt** is a declared **float** variable and **chr** is a declared **char** type variable, then the following conditional expression will yield a **double** data type value regardless of the truth of expression1:

expression 1 **? chr : flt**

The **float** variable, **flt**, will automatically be converted to a **double**. The character data type variable **chr** must then be promoted to be compatible with the now double precision floating point value represented by the variable **flt**. The concept of short circuit evaluation does not conflict with that of

type conversion. This is due to the fact that the data type of an expression can be determined without numeric evaluation.

Primary Operators

Primary Expressions

The alert reader should have noticed that many operators were said to require "an expression that evaluates to an [extended] fundamental data type or pointer. Primary expressions are those basic tokens capable of representing numeric data. A complete list of C's primary expressions follows:

constant
string
variable identifier
($_{opt}$expression)$_{opt}$
primary-expression [expression]
primary-expression (expression-list$_{opt}$)
primary-lvalue . identifier
primary-expression -> identifier

The following term definitions apply:

expression:

($_{opt}$ |unary optr |$_{opt}$ |operand | |binary optr | |unary optr |$_{opt}$ |prim-ary exp | etc.)$_{opt}$

The | symbol is used here as a delimiter. The subscript $_{opt}$ denotes optional tokens. The string optr is an abbreviation for operator. Optional parentheses used to delimit subexpressions may also be present.

expression list:

$$(_{opt} \text{ } expression \text{ })_{opt}$$

or $(_{opt}$ *expression* $)_{opt}$, $(_{opt}$ *expression* $)_{opt}$, etc

An expression list is comprised of one or more expressions, each optionally delimited by a matched pair of parentheses and separated by commas.

Constants are the simplest primary expressions, as they literally represent a data value. Constants were detailed in the last chapter. Strings are a special form of constants which will be detailed in chapter 9. String constants are composed of one or more character values delimited by a matched pair of double quotes. An identifier is nothing more than a variable name. Variables must be explicitly declared before they can be assigned values.

Expressions are comprised of one or more primary expressions joined via binary or ternary operators. Unary operators and parentheses may optionally be included. Correct syntax must, of course, be observed. The other primary expressions will be detailed under the relevant ensuing sections and in subsequent chapters.

The Parentheses (Argument/Associative) Operator

Operator	Description	Syntax
( )	associative	*(expression)*
	argument	*primary expression* (*expression list*$_{opt}$)

NOTES

- The precedence and associativity of both operators is equivalent.
- The parentheses operator will accept any expressions that can be resolved to an [extended] fundamental data type or pointer value.

The primary parentheses operator really serves two purposes: it coerces associativity and it delimits the argument lists of functions. (Also, by identifying the argument list of a function, the parentheses pair identifies function headers and calls.) In the latter use, the parentheses serve a role of punctuation, as a matched pair is used to enclose all actual or formal arguments. This readily enables the compiler to differentiate between arguments and subsequent tokens.

Notice how the lack of punctuation creates considerable ambiguity in the following expression:

mddle squrt k , i + 15, j – 3.145 * b

The argument operator can be positioned in a number of ways while still satisfying syntax requirements. The following expression illustrates one such positioning scheme:

mddle (squrt (k), 1 + 15, j – 3.145) * b

One must also satisfy argument matching criteria. More will be said on functions in chapter 6.

Pairs of parentheses may also be used to explicitly define associativity. The reader should be familiar with this concept because such usage is clearly outlined in the field of algebra. The associative operator is a very handy tool in that it allows the programmer to code without the extensive use of explicit temporary variables. For instance, the following three statements:

r = a I b;

s = r + c;

t = s * d;

can be written as one statement with the aid of the associative operator:

t = (s = (r = a I b) + c) * d;

In this example, the three pairs of parentheses override the normal precedence of the binary operators, as the operations occur from order of lowest to highest precedence. Assuming that the intermediate variables are only needed as explicit temporaries, this statement can be simplified to:

t = ((a I b) + c) * d;

Remember that although parentheses force associativity, they have no effect on order of evaluation for commutative operators. Consider the following statement:

heat = (squ(1.50 + a) + log(b)) + bnv(1,m,n);

Despite what the associative operator pair implies and without further system knowledge, one cannot ascertain the order of evaluation of these three functions.

The associative operator is also used quite often within derived data types to coerce stronger binding of primary and unary operators. For example, the following two declarations describe two totally different derived data types:

int (*ptr_ary)[5];   vs   **int *ary_ptr[5];**

The left-hand declaration defines the identifier **ptr_ary** to be a pointer to an array of five integers. The parentheses were necessary to bind the indirection operator, *****, tighter to the identifier in preference over the subscript operator. Without these parentheses, the declaration would define an array of five pointers to integers, as in the right-hand statement. This second declaration could also have been coded as:

int * (ary_ptr[5]);

When used as associative operators on data types (i.e. non-literal primary expressions), parentheses can appear on the left side of an assignment expression.

Outside of argument and associative uses, a pair of parentheses may only be used in control flow statements (chapter 5). Here it serves a purpose comparable to its argument capacity, namely to delimit the object subexpressions of the control flow statement. The reader may find it easier to think of the parentheses as punctuation when used around function arguments and control flow statement capacities, but not when used as the associative operator.

The Subscript Operator

Operator	Description	Syntax
[ ]	subscript	*primary expression* **[***int-expression***]**

NOTES

- The primary expression operand for the subscript operator must resolve to an array name or indirection operator/pointer referencing the first element of the array.
- The expression operand(s) must resolve to an integer value and must be in the range of array elements (i.e. can take on numeric values from zero to width-1) for referencing array elements.
- Arrays are filled and referenced row by row.
- Arrays may not be of register storage class.

The subscript operator is used to both identify arrays and reference specific elements in those arrays. For an n-dimensional array, the notation is as follows:

data type name   array identifier **[***m***] [***n***]...[***r***]**

The bracketed letters represent expressions which can be resolved to integer values (primary expression). These integer values, termed subscripts, denote through their number the dimensionality (or rank) of the array, while their actual values define the width of each dimension. For example, the following

statement declares **ary** to be a two dimensional integer array of row length (first subscript) 5, column height 2 (second subscript), and width 7 (third subscript):

<div align="center">

int ary[5][2][7] ;

</div>

The array **ary** is said to be a 5x2x7 array. It has a dimensionality or rank of three and a total number of elements of seventy.

Outside the declaration statement, subscripts denote specific array elements in the named array. Array elements are numbered from zero to the subscript minus one. Therefore, the notation **month[2]** denotes the third element of this array. Suppose the value of the seventieth element (the last element) of the above array is to be doubled. This can be directly accomplished with the following statement:

<div align="center">

ary[4][1][6] *= 2;

</div>

Probably the most confusing attribute of arrays is the fact that in C, the subscript n in a declaration refers to n elements, while throughout the rest of the program n refers to the $(n+1)$th element because subscripting begins at zero. Arrays are discussed in greater detail in chapter 9.

The Member Access and Selection Operators

Operator	Description	Syntax
.	member access (dot)	$(_{opt}$ *primary-lvalue* $)_{opt}$.$(_{opt}$ *identifier* $)_{opt}$
->	member selection (arrow)	$(_{opt}$ *primary expression* $)_{opt}$ -> $(_{opt}$ *identifier* $)_{opt}$

NOTES

- The primary lvalue operand for the dot operator *should* resolve to an lvalue naming a structure or union (i.e. a structure or union name).

- The primary expression operand for the arrow operator *should* resolve to a pointer to a structure or union.
- The identifier operand for both operators must name a member of the structure or union.
- The result of both operations is an lvalue referencing the member of the structure or union named by the identifier.

The dot operator is the conventional method for referencing a member of a structure or a union (chapter 10). For example, suppose the structure declarator **time** was declared as:

```
struct time {
long int year;
int month;
int day;
int hour;
int minute;
int second; }
```

and an identifier **event** was declared to be this type of structure by the statement:

```
struct time event;
```

(Unlike array names, structure names are actual lvalues in C.) Then, a member of the time-flavored structure **event** could be accessed as an lvalue using the dot operator. For example, the statement:

```
event.hour = 13;
```

assigns a value of 13 to integer variable **hour**, which is a member of the structure **event**. The member identifier must be a valid member of the specified structure.

The member selection or arrow operator was included in C as a convenience, because like array notation, its effect can be obtained through the use of the indirection operator. Normally the arrow operator is used to set a pointer to a member of a structure. The statement:

struct time *ptr_time;

declares **ptr_time** as a pointer to time-flavored structures. The following statement associates this pointer variable with the structure **event**:

ptr_time = &event;

Then a member can be accessed with the arrow operator, as through the expression:

ptr_time -> hour

or through the equivalent expression:

(*ptr.time).hour

Constant Expressions

A constant expression is an expression that always resolves to a single, invariant value. There are several instances where C demands constant expressions: as the assigned value during initialization (i.e. the initializer) of external and static variables (chapter 7); after the keyword case in switch conditional statements (chapter 5), and as array bounds (chapter 9). In the latter two cases, the constant expression can consist of integer and/or character constants (including escape sequences), and sizeof expressions optionally in conjunction with the unary minus, -, or bitwise NOT, ~, operators. If more

than one constant and/or sizeof expression is present, each must be connected to the next by either the ternary conditional operator; ?:, or one of the following binary operators.

$$\| \quad \&\& \quad | \quad \wedge \quad \& \quad == \quad != \quad < \quad > \quad >= \quad => \quad >> \quad << \quad = \quad + \quad - \quad * \quad / \quad \%$$

Subexpressions may be defined by the use of parentheses. Therefore, the syntax for a constant expression is as follows:

(opt(optconstant BINOP constant)opt BINOP constant)opt

and/or

(opt(optconstant expression)opt ? (optconstant expression)opt :
    (optconstant expresion)opt)opt

and/or

(optsizeof(expression or data type name))opt

where BINOP is symbolic of one of the previously listed binary operators.

Initializers of external or static variables may also contain the unary "address of" operator, &, if applied to previously defined external or static identifiers. Automatic and register variable initializations can also include previously initialized variables of appropriate scope and calls to previously defined functions (chapter 7).

Data Type Conversions

When a statement or expression contains operands of mixed data type, they are necessarily converted to a common type according to the rules described in this section. The reader should recall from chapter 3 that in an expression or during function argument passing, the following conversions automatically take place:

- All primary values of char or short (int) data class are promoted to int fundamental data class.
- All primary values of float are converted to double fundamental data class.

A second type of conversion that has been mentioned is that performed by the logical, relational, and equality operators. Namely, regardless of valid operand data type, these operations yield a value of int 1 for a true outcome and a value of int 0 for false. These returned values can be used in an expression as can any other numerical value. This is shown in the following statement:

<div align="center">

capflag = ltr >= 'A' && ltr <= 'Z' ;

</div>

The variable **capflag** will be set to one if the variable **ltr** is a capital letter.

Thirdly, when one of the operands of a binary operator is of mixed type, the lower is normally promoted to the wider data type as defined by the widening hierarchy (q.v. chapter 3). There are three exceptions to this rule:

- The automatic conversion of char and unsigned int to int and of float to double.
- Some operators will not allow certain data types (e.g. the bitwise operators required integral operands).
- During assignment, the value on the right will be converted to coincide with the lvalue data type on the left.

The rules for conversion of mixed data type operands can be stated as follows:

- char and short (int) will automatically be converted to int, and float will automatically be converted to double.
- Then, if either operand of a binary operator is a double, the other operand will be converted to double. The result will be a double.

- Otherwise, if either operand is a long, then the other will be converted to a long. The result will be a long.
- Otherwise, if either operand is an unsigned int, then the other will be converted to an unsigned int. The result will be a unsigned int.
- Otherwise, all that is left are int's, and the result will be an int value.

The particulars of this process are similar to the process of type conversion across an assignment (see chapter 3, "Type Conversions [Advanced])". It differs only in the fact that all int, short, and float data types are automatically promoted and remain promoted throughout evaluation.

The programmer must be careful to code expressions correctly. For example, if an expression was required to divide a value of decimal seven by five and add a value of Pi, the following statement would be inadequate,

$$7 / 5 + 3.1415$$

because 7/5 has two int operands and the result would be int 1. Note that the proximity of the floating point number did not affect this outcome, because the division operation was performed before the addition. To obtain the algebraically correct value, one of the following subexpressions must be used in lieu of this integer division:

$$7.0 / 5.0 \quad or \quad 7.0 / 5 \quad or \quad 7 / 5.0$$

The nonsignificant zeroes are optional. Of these three subexpressions, the first is preferable because it saves execute time which would otherwise be used for processing type conversions.

Finally, conversions may be coerced through the use of the cast operator. The use of the cast operator is equivalent to the use of an explicit temporary of appropriate data type. For example, the following two sections of code are equivalent:

```
section one                    section two
int m;                         int m;
double db1, db2;               double db2;
m = (expression);              m = (expression);
db1 = m;                       db2 = squrt ((double)m);
db2 = squrt(db1);
```

> where squrt is a utility function requiring a double argument and is defined elsewhere

The subexpression **(double)m** in section two forces the copied value of integer variable **m** into a double precision floating point format. If one assumes that the variable **m** is not needed after this program section (i.e. **m**, itself, is a temporary variable), then the following section of code would be sufficient:

```
double db2 ;
db2 = squrt((double) (expression)) ;
```

In regard to type conversions, it is important to recognize that the actual values of variables remain unchanged except when an assignment occurs.

```
int a = 5;
float b = 3.1415;
b = a / .333;
```

Although the value of **a,** integer 5, will be converted to double format for the division in line 3, subsequent use of variable **a** will confirm that the stored value remains int 5. The computer only makes a copy of a variable for its use unless instructed otherwise, such as through an assignment statement. In any instance, the data type of a variable can never change.

Operational Statements

Chapter 2 presented a basic approach to the construction of operational statements. In this chapter, C's operators were presented, and a explanation of lvalues and primary expres-

sions was given. The question of what constitutes an allowable operational statement in C still remains.

C is a very permissive language. It has been shown that there is a great deal of flexibility in source code format and data type usage. A similar condition exists for statement composition. For example, C allows "useless" statements such as the following (assuming **d** has been declared and initialized):

$$(d * 3) + 15.123 ;$$

If the statement is included in a program, it will cause the defined operations to occur. However, since nothing is done with the resultant value, it is effectively lost. In other words, this statement does nothing except waste time and change the values in certain registers of the CPU.

The reason C allows such wasteful coding and allows overflow and underflow errors involves the philosophy of compactness and flexibility. Generally the more capability a programmer designs into a package, the larger and more complex (expensive), and paradoxically, sometimes more restrictive that product will be. C leaves the responsibility for many aspects of proper program design up to the programmer, although utility functions and programs can and have been developed to assist the programmer.

This discussion leads directly to the question "How do we retain the value of an expression?". The obvious answer to this is "by storing it in memory". We can explicitly store a value in specific areas of memory by using pointers as shown in the following example:

```
char * ptr ;
ptr = 6321        /*hazardous*/
*ptr = 16
```

For the reader unfamiliar with pointer operation, this sequence of statements assigns a value of 16 to the memory location

6321. *Use of pointers in this manner is extremely dangerous* as normally the programmer is not aware of the details of the operating environment. Let us expound on this idea.

The operating system consists of the control program and associated hardware that supervises and controls essential internal process such as system error handling, I/O, job control, and most importantly to this discussion, memory allocation. Different sections of a computer's RAM are used for different purposes depending, in part, on the type of job at hand. Each area of memory is patrolled and managed by the operating system. The trouble with the former program section is that the operating system has not been informed that memory location 6321 is being used by the program to store the value 16. If this location is allocated to some other purpose, either new information may be "written over" the value of 16, or this value will replace some other possibly valuable piece of information, or possibly both may occur.

It should be obvious to the reader that the answer to these worries lies in allowing the operating system to do its job without intervention. This can be accomplished by assigning the value returned by an expression to a variable (i.e. assigning a variable an rvalue). The operating system assigns a variable the appropriate storage block in memory when said variable is declared (i.e. assigns the variable an explicit lvalue). In the preceding character pointer example, the variable pointer, **ptr**, underwent this treatment. Direct memory assignment statements are illegal in C.

```
*6321 = 16 ;       /*illegal*/
&ptr = 2000 ;      /*illegal*/
```

Therefore, for an operational statement to be useful, it must generally be assigned to a variable. However C, unlike most other languages, allows the very liberal use of the assign-

ment operation. This operation can be used much like any other (sub)expression, although parentheses are normally required to enforce proper association. Examine the following examples:

```
int a=1, b, c=a*31 ;

m=n=stor=PIE*PIE ;

ast = dif * (sub1=412/LOW) ;

ttime = max(( pause=tim1*tim2),tim3 ) ;

while(( i=1 ) > stop )
```

The first statement illustrates declaration/initialization hybridization, a topic discussed towards the end of chapter 3. The second statement demonstrates an "equalization" multiple assignment. This sort of statement is allowed in a considerable number of other languages. The third example is an operational/assignment statement containing or "nesting" a comparable subexpression. This sort of statement is usually disallowed in other languages. The fourth example demonstrates the use of an assignment expression within a function call/assignment statement. The fifth exemplifies a while control expression (chapter 5), nesting an assignment statement.

Keep in mind that despite its flexibility, C still includes syntactic, punctuation, and use rules that must be obeyed. The following statement;

```
a + 1 = b * 3 ;    /*illegal*/
```

violates the syntax model for assignment statements:

lvalue = expression

and is consequently disallowed. The preceding discussion(s) also holds true for the ten compound assignment operators.

If the reader has problems interpreting these lines, then it might be initially helpful to mentally separate such lines into two simple expressions. For instance, the fourth example statement could be written more clearly as:

```
pause = tim1 * tim2 ;
ttime = max(pause, tim3) ;
```

Perplexing code becomes much more easily readable with the passage of time and effort.

5

Control Flow and Program Development

Introduction

To this point, the programs featured in this text have been simplistic in nature. They have not accepted input, and their execution pattern has been sequential in nature. Such programs have very little practical value aside from logo, header, error, or other invariant message printing. In this chapter, an attribute that is an innate part of C or any computer language,

control flow, will be discussed. Control flow refers to the ability of a statement to influence the sequence of program execution. Sequential statements represent the most basic control flow statements. We have used sequential statements in the preceding chapters. When control is passed to such statements, they are executed once. Control then passes to the next statement.

Three new control categories will be introduced in this chapter: decision or conditional statements, iterative or loop statements, and the goto statement.

In computer science the word *construction* denotes the method of logically organizing statements to produce the desired control flow. Control flow constructions enable us to specify which statements are to be executed given a predetermined set of conditions. C's control flow constructions are largely ordinary relative to other languages, aside from the concise syntax and the lack of restrictions on their use.

This chapter also advances the standard C library **scanf()** function as an input utility. The scanf() function corresponds to the printf() output function utility, as it uses much the same format and conversion specifications as printf(). Input and control flow capabilities serve vital and irreplaceable purposes that greatly extend the capability of any programming language.

THE IMPORTANCE OF DECISION-MAKING

Decision-making ability is a trait that gives an entity flexibility. Without this attribute, an object is a simple machine, incapable of any change of operation. For example, a machine press that punches a part out of stock material every five seconds while operating properly is devoid of any decision-making ability. However if a simple trip mechanism is installed that stops the compression stroke when no stock material is

present between the die (mold) walls, then this machine has been imbued with decision-making ability.

Some might argue that the aforementioned press is incapable of decision making because it is non-intelligent. Such a view clouds the concept of decision-making with intelligence. Few would argue that this machine is intelligent. The press can correctly be described as a decision-making entity by way of the fact that *a stimulus causes a change in its routine behavior*.

For a functioning computer, stimuli is in the form of binary data. These data values can be produced internally, from values contained in the program at compilation time, from values derived from some aspect of the operating system, or externally from some digital input device, usually a keyboard or secondary storage device.

One of a computer's basic features is its ability to compare two numbers. Specifically a computer can judge whether a number is greater than ($>$), equal to (==), less than ($<$), greater than or equal to ($>$=), or less than or equal to ($<$=) another such number. These three comparisons then, are the only decisions most computers can make. All other higher level decision-making abilities are built on these.

General Problem Solving Techniques

Because computers have powers limited to binary data I/O, storage, manipulation, and comparison, the programmer must present a problem to the computer in an acceptable form. Knowledge of the facts of operation of a computer are generally not a prerequisite to programming, as high-level programming languages such as C are available for implementing the solution. The question still remains, however, as to how to translate an actual practical problem into an acceptable C program. This problem solving process can be divided into six steps.

Step 1 Problem definition. Before a problem can be logically tackled the programmer must fully understand the nature of the problem — "what exactly do I want the computer to do?" Without the comprehension and definition of the problem at hand, proper program design often turns into a hit-or-miss proposition.

Obviously computers are only qualified to deal with a specific class of problems. Computers can handle problems that deal only with data that can be logically represented numerically and whose solution may be obtained through "arithmetic", boolean, and/or value comparison operations. As a result, computers are not capable of making valid subjective decisions or performing tasks that require true intelligence (e.g. questions such as "who is the better political candidate?" or "how should I go about solving this problem?")

Step 2 Algorithm development. Once a problem has been properly defined, a detailed, finite, step-by-step procedure for solving it must be developed. This procedure is known as an *algorithm*. The most common means of initially defining an algorithm is through the use of *flowcharting* techniques (appendix E). This process is probably the most difficult to describe or teach because it involves concepts of logic and creativity.

Inductive reasoning should be applied to the flowcharting process. That is, the programmer generally should reason a problem from the particular to the general case. This text's programs have been extremely inflexible up to this point because they have dealt only with very specific and unvarying data values and operations. The algorithm development step is ordinarily the most difficult because most people are not accustomed to proceeding in discreet, incremental steps; instead they tend to unconsciously or intuitively jump to a solution. The advancement of the former method is essential as it is the basis of algorithm development, which in turn is considered *the key* step in problem solving using a computer.

Step 3 Algorithm source coding The algorithm developed in step 2 must be translated into a set of instructions that a computer can understand. These instructions, along with additional qualifying format and other code comprise a computer language.

The specific computer language used may affect this process. Certain programming techniques, operations, and representations are encouraged by certain languages, while in others, they are more difficult or impossible to implement. C is a very general language, but it may not be the most convenient for many specific classes of problems. Often an intermediate process is followed to bridge steps 2 and 3.

An algorithm in flowchart form may be written as an English translation of the necessary computer instructions. This intermediate representation of a algorithm is termed *pseudocode*. Pseudocoding is a convenient tool for helping inexperienced programmers code as well as for documentation. However many programmers shun pseudocode because it is time-consuming and, where external, involved documentation is unwarranted, unnecessary.

Step 4 Symbolic binary representation. This is an impressive name for the process of "typing in" the program. Alphanumeric symbols are encoded into their ASCII or other binary pattern. Older facilities incorporate card punch/reader or teletype I/O devices, while most current systems have one or more keyboard/video display units. Large amounts of time can be saved using editing packages which are available for the latter devices. The coded algorithm is referred to as the source program. It is generally stored on a peripheral memory medium as a source file.

Step 5 Compilation and program execution. The source code from step 4 must be translated into a form understandable by the CPU — machine language or object code. C is a compiled language. Compilation is a translation of the source code into a

machine language or object code. This can be saved as an object file. One or more object files comprise an executable program. More information on this step will be presented in chapter 8.

Step 6 Debugging. A "bug" is an error in a program. There are three kinds of errors that can be encounted in the computer problem solving process.

- *Compilation time errors* are associated with the failure of a source program to compile correctly because of syntactic or usage errors. Most compilers also give warning messages to indicate legal but possibly hazardous coding practices or constructions.

- *Run time errors* result from ambiguous or illegal conditions that arise during the execution of a program. While C has neither overflow nor underflow error messages, illegal operation conditions such as referencing an out-of-range array element or data type mismatch conditions such as the use of non-integral operands in bit-wise operations are likely to result in run time errors. C and C compilers generally have much less strict policing of run time conditions than other languages.

- *Logical errors* do not show up as compiler-generated error messages. Rather program output (or system operation for system programs) deviates from the expected. These errors are generally associated with source code design.

Compilation time errors abort the compilation process either immediately or following the compilation phase in which they occur. No object code is produced. These errors result from syntax errors in the source code or in the command language responses necessary for compilation. An example of these are missing statement delimiters — ; and mistyping the source file name, respectively. Normally program syntax errors simply require returning to step 4 for error correction.

Some compilers also produce warning messages when unusual conditions are present that are acceptable but unorthodox. The programmer then has the option of ignoring the warning or investigating its cause.

Run time errors may be caused by a number of factors such as incorrect input, syntax errors, incorrect control flow, data type mismatch, etc. Identification of the error will typically suggest the correct solution. Normally run time errors are associated with coding inadequacies.

Logical errors can be loosely divided into two categories: the blatant errors and the true logical errors. Blatant errors involve the substitution of an incorrect variable, operator, or keyword in the place of the correct one. Often such an error is an oversight or entry error.

True logical errors point to failings in steps 1, 2, or 3. When a program is coded as intended but does not perform adequately, a fault exists in the definition of the problem, in the algorithm designed to solve it, or in the language coding translation of the algorithm's intention. The "hidden bug" is a special instance of this type of error, and arises when only certain input values cause output irregularities. A program can run correctly for months or years until it is presented with an unusual value such as an inordinarily large or small value, zero or a negative number. By this time the program is usually unsupervised, and the error may never be detected. Sometimes run time and logical errors can only be discovered by mentally following through a program and "executing" it on a line by line basis as if one were the actual computer.

Flowcharting and Pseudocoding

Analyzing a problem in an orderly, logical fashion is generally the most efficient means of attacking it. One useful approach is to view any problem solving method as a series of decisions and actions connected in a well-defined manner.

Such an approach is widely advocated for program design and can be implemented through flowcharting.

Flowchart symbols, as described in appendix E, are used to define the order of the steps necessary to solve a problem. For instance, the flowchart representation of the steps needed to solve the problem "Should I set my alarm tomorrow?" would take the form presented in figure 5.1.
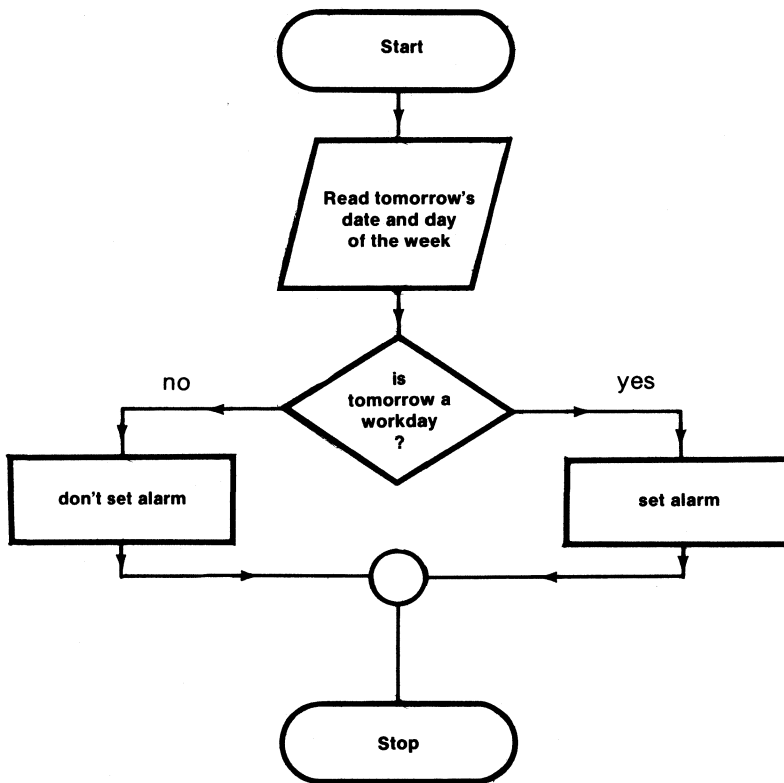
FIGURE 5.1. Flowchart for deciding when to set an alarm

In constructing this flowchart it was implictly assumed that the alarm need only be set on workdays, and that it would automatically be set to the correct time. The process of problem solving actually occurs during the development of an appropriate algorithm, which is represented by the flowchart in the example. A computer therefore, does not actually solve problems, it merely follows the steps defined by the programmer.

Flowchart execution should begin at the top of the diagram and procede to the bottom of the diagram. The first step in any program is a "start" terminal operation which denotes the loading and initial execution of a program. The next step in the flowchart depicted in figure 5.1 is an input operation that reads in the next day's date and day of the week. Notice that this step is correctly represented by the I/O program symbol. This information must then be compared with a set of predefined criteria to determine if the next day is a workday. If the date is determined to be a workday, the alarm will be set. If not, then the alarm will not be set. If not setting the alarm entailed no action whatsoever, the left process block could be deleted. Both branches of the decision step would then join at the collection node and continue to the stop terminal. C programs almost always stop at the last closing brace of main().

Once a viable flowchart has been created, an algorithm may then be converted directly to source code or indirectly so by way of pseudocode.

CONTROL STRUCTURES

There are three basic types of *control structures* that are used in C, as in most computer languages. It has been demonstrated that any algorithm capable of being coded may be written using these three types of control structures —

namely sequential execution, conditional execution (also called decision or selection), and iterative (or loop) execution*.

- **Sequence** — the serial execution of one statement immediately after its predecessor. This structure is represented in flow diagrams by an arrowed line connecting the two processes.
- **Decision** — the execution of either one section of object code or another. This results in the control flow branching. The actual branch taken is determined by the evaluation of a *test condition*. This structure is indicated by a decision diamond containing two paths. One path indicates true (yes), while the other indicates false (no). These two paths usually join at the collector node.
- **Iteration** — this structure allows execution of the same section of code any integral number of times (including zero or just one time). This structure is indicated by the routing of flow back to a previous step.

PSEUDOCODE

Sometimes it is desirable to translate an algorithm to an intermediate form, between that of a flowchart and the source code. Pseudocode is an English approximation of source code that follows the rules, style, and format of C but ignores most punctuation. Pseudocode substitutes the English equivalent for some operations. Our alarm setting algorithm may be pseudocoded as follows for a mechanical clock,

---

* C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules"

```
main()
  { integer date ;
      char day ;
      read in date and day ;
      if day is not saturday or sunday
        and date is not a holiday ;
        then { set hour hand to eight ;
                set minute hand to six;
                set alarm button to "on" ;
              }
      else   do nothing ;
  }
```

Figure 5.2. Pseudocode for deciding when and how to
               set an alarm.

Compare this figure to figure 5.1. The flowchart gives a much clearer image of the control flow of the program indicating decisions, actions, and repetitions. Pseudocode, on the other hand, defines conditions and actions more closely. For instance, the procedure for setting the alarm is detailed in full. The decision step is also more fully formalized although the conditions which would make variable **day** a holiday are not spelled out.

Reference Problem

The major concepts of this chapter will be developed with the aid of the following problem. A teacher wants to design a program to read in a list of final numeric scores and on the basis of these assign a letter grade. The following grading scheme is observed:

Score	Grade
90 - 100	A
80 - <90	B
70 - <80	C
60 - <70	D
0 - <60	E

Furthermore, some members of this class audit the course. It is originally assumed that auditing members of the class do not receive "real" test scores and that the program need only handle one set of input values at a time. These two limiting assumptions will be relaxed and additional simple numeric manipulation will be included later in this chapter.

The Conditional If Construction

The **if** statement is the most extensively used conditional statement in C as well as a number of other languages. The syntax of an **if** statement is:

If (*expression*)
statement

The expression after the **if** keyword is referred to as "the object expression," "the test condition," "the test" or "the condition". The statement following the expression is referred to as the *object statement*. It is also referred to as the substatement.

During the execution of the if statement, the parenthesized expression is evaluated first. If it is true (evaluates to a non-zero value), then the object statement is executed. If it evaluates to false (zero), then the object statement is ignored and control passes to the next statement.

Ordinarily the object expression involves an equality or relational test. For example, the reference problem's decision steps could be written as shown below:

```
if (score >= 0)
    grade = 'E';
if (score > 60)
    grade = 'D';
if (score > 70)
    grade = 'C';
if (score > 80)
    grade = 'B';
if (score > 90)
    grade = 'A';
if (score == 666)
    grade = 'T';
```

The grade **T** denotes auditing students, and the program section assumes that auditing students were assigned a score of 666 to differentiate them from the other students. The direct assignment of **T** to the variable **score** would cause ambiguity because the ASCII value of decimal 84 is associated with this character. A value of 84 can also represent a grade of B. Remember, in C the integer and character classes are very similar.

Note that not only is **score** involved in six relational tests but that during execution of this one section of code, the variable **grade** could be assigned a character value then reassigned another character value. As a case in point, consider a value for **score** of 74. Since this value is greater than zero, **grade** is initially assigned a character value of **E** during the first if decision construction. Flow then passes into the second if statement where **score** is found to be greater than 60. **grade** is then reassigned a value of character **D**. Control flow then passes to the third if construction where **grade** is assigned a value of **C**. After this statement, **score** no longer causes the remaining condition expressions to evaluate to true so no further reassignments occur.

Multiple if statements should not be used in this manner for four reasons:

- Execution time is wasted executing unnecessary object statements.
- Execution time is wasted evaluating redundant multiple relational and/or equality expressions.
- This construction is only applicable to steadily increasing or decreasing categories and then only if the object statement is reversible (i.e. a printf() would be irreversible).
- Multiple if constructions (used on non-mutually exclusive categories — see below) are confusing and highly error prone.

This type of construction should only be used for mutually exclusive categories. Category B is mutually exclusive of category C if the inclusion of an item in C precludes it from being a member of B (and vice-versa). As it stands, the example does not contain mutually exclusive categories. A score can "trip" (or cause a yes value) in more than one if statement.

THE CONCEPT OF FLAGS

Often a programmer wants a statement to be conditional on some problem parameter which is not associated directly with calculations. In this type of situation, dummy variables known as *flags* are often used with the if conditional statement. A flag can be set to any one of a variety of known values, each corresponding to one of a variety of different problem parameters.

Suppose that separate equations must be used to calculate drag on a object traveling in a forward versus a backward direction. The following section of code uses flags in combination with conditional if statements to branch program execution depending on the sign of the velocity. A negative velocity indicates a "backward" motion.

```
int dr_fl;
float drag, vel, other parameters;
    ...
if (vel > 0)
    dr_fl = 'f';
if (vel < 0)
    dr_fl = 'b';
    ...
if (dr_fl == 'f')
    drag = forward drag equation;
if (dr_fl == 'b')
    drag = backward drag equation;
```

The advantage of flags may not be obvious from this example because the four if statements could have been coded more succinctly as:

```
if (vel > 0)
    drag = forward drag equation;
if (vel < 0)
    drag = backward drag equation ;
```

The value of flags is more obvious when a number of parameters influence the flag setting or when a flag can assume any one of a number of different settings at different points in the program. Flags are also useful in that they may hold dummy values indicative of a condition even after the actual numeric values of that condition cease to exist in the program. Flag arrays are especially suited for this sort of past value indicator. Flags are always mutually exclusive by nature so they may be handled correctly by multiple if constructions.

The if-else Statement

The simple if statement described over the last few pages is actually a simplification of its parent statement, the **if-else**. The syntax of an if-else statement is:

If (expression)
      statement1
else
      statement2

As before, when the **if** object expression evaluates as true, statement1 is executed, and the **else** section is ignored. When the **if** object expression evaluates as false, statement1 is ignored, and statement2 is executed. The **else** section is optional. Without it, however, construction reverts back into the simple **if** construction.*

After either the **else** branch (execution of statement2) or the **if** branch (execution of statement1), control passes to the next statement. The exception occurs when a **goto** is executed in one of the object statements, as explained later in this chapter.

if-else constructions are particularly well suited for non-mutually exclusive categories. If this construction had been used for our reference problem, the decision could have been written as shown on page 191. Compare this figure to the decision section using the simple **if** code section presented on page 189.

The final **else** clause assigns a value of R to the variable **score** when the grade is out of bounds — greater than 100 or less than zero. This process is called *data validation*.

---

* The else branch may also be effectively bypassed by replacing statement2 with a semicolon. Null statements are always valid.

```
if (score == 666)
    grade = 'T';
else
    if (score > 100)
        grade = 'R';         /* 5 */
    else
        if (score > 90)
            grade = 'A';
        else
            if (score > 80)     /* 10 */
                grade = 'B';
            else
                if (score > 70)
                    grade = 'C';
                else                    /* 15 */
                    if (score > 60)
                        grade = 'D';
                    else
                        if (score >= 0)
                            grade = 'E';
                        else
                            grade = 'R';
```

NESTED STATEMENTS

Note in the preceding example that the object statements in most of the **else** expressions are themselves **if-else** constructions. In other words, this whole section of code is actually one statement that is comprised of seven hierarchically nested **if-else** "chain" (sub)constructions.

If, at any time during the general top-to-bottom execution of these expressions, an **if** expression evaluates to true, then the object statement of that **if** is executed, and control flow is passed to the statement immediately following the entire nested chain. For example, consider once more a score of 74. The first **if** object expression, **score == 666**, would obviously evaluate to false. Control would then pass to the **else** object

statement. This statement is another **if-else** construction which is then executed as any other statement would be. The value held by **score** is not greater than 100 so again the **else** clause is branched to. Execution of this object statement involves another **if-else** object statement. This chain process continues until the **if** expression on line 13 is encountered. The condition (**score** > 70) evaluates as true. In this case, therefore, the object instruction of the **else** clause is not executed, but rather the **if** object statement is executed. Specifically, line 14 assigns the character value **C** to the variable **score**. Chain execution then ends.

Note that although no braces explicitly define the hierarchical structure of this section of code, such a definite structure exists. In fact, each succeeding **if-else** statement is an object statement (or "substatement") of the one preceding it. This section of code includes seven **if-else** statements. Seven corresponding hierarchical levels are included-six substatements and one inclusive, parent **if-else** construction.

The else-if Variation

The indentation scheme presented in the previous section is technically correct. Since this construction is used frequently to represent nested decisions, and since lengthy nested constructions have a way of running off the "page", a variation of the **if-else** construction, termed the **else-if** construction, is often used.

The decision code section presented on page 191 is represented by the **else-if** construction presented on the next page. This construction involves nothing new; it only utilizes the

freeform nature of C to restructure format. The section begins with an if expression and is followed by an else expression containing a chain of six nested if-else's.

```
if (score == 666)
    grade = 'T';
else if (score > 100)
        grade = 'R';
else if (score > 90)
        grade = 'A';
else if (score > 80)
        grade = 'B';
else if (score > 70)
        grade = 'C';
else if (score > 60)
        grade = 'D';
else if (score >= 0)
        grade = 'E';
    else
        grade = 'R';
```

In addition to the space savings, the else-if construction is more appealing to the eye. All the nested object statements and instructions are aligned beneath one another making it easier to locate a particular comparison or assignment. The else-if construction is generally used in preference over the if-else when two or more of these nested statements are included. The reader should keep in mind that although the indentation does not signify it, the else-if construction is a hierarchically nested structure.

Compound Object Statements

All the conditional statements up to this point have contained only one object statement. More than one object statement, surrounded by a pair of braces, may be substituted for a lone object statement. The braces and the enclosed substatements define a block. Because this block substitutes for a single object statement, and yet obviously contains many individual statements, it is termed a *compound statement*.

Syntax for the general **if-else** construction can now be written as:

```
If (expression)
  { statement1a
    statement1b
    ...
  }
else
  { statement2a
    statement2b
    ...
  }
```

Compound statements contain a semicolon after each simple statement. As a simple example, suppose the program was required to output the total quality points (e.g. a grade of B is three quality points). An additional instruction per decision would be required as shown below:

```
If (score > 90)
  { grade = 'A';
    t_qu_pts +=4;
  }
else
  ...
```

For a score of 94, the character value **A** is assigned to the variable **grade** and integer **4** is added to the variable **t_qu_pts**. Compound statements may be used in any of the conditional or loop statements.

Nested Simple If Statements

Occasionally the reader may run into a construction such as:

```
if ( expression1 )
   {statement1a
      . . .
      if (expression2)
         {statement2a
            . . .
            if (expression3)
               {statement3a
                  . . .
               }
         statement2k
            . . .
         }
   statement1k
      . . .
}
```

In order for a nested substatement to be executed, *all* of the test conditions of the **if**'s under which this statement is nested must be true. For example, statement3a is only executed if expressions 1, 2, and 3 all evaluate to true. On the other hand, although statement2k is physically located after the innermost **if**, it is actually associated with the second **if** statement. Its execution, therefore, only depends on expressions 1 and 2. This type of construction can be used to define mutually exclusive sets.

As presented below, the simple **if** decision section example

can be rewritten using nested **if** constructions to obtain mutually exclusive conditions. Because each comparison is mutually exclusive, the **if** parent statements (which are delimited by blanks lines in the figure) may be arranged in any order. Error conditions could similarly have been included using nested simple **if**'s.

```
if (score >= 0)
    if (score <= 60)
        grade = 'E';

if (score > 60)                /* 5 */
    if (score <=70)
        grade = 'D';

if (score > 70)
    if (score <= 80)           /* 10 */
        grade = 'C';

if (score > 80)
    if (score <= 90)
        grade = 'B';           /* 15 */

if (score > 90)
    if (score <=100)
        grade = 'A';
                               /* 20 */
if (score == 666)
    grade = 'T';
```

Nested **if** statements are rarely used in this particular way because constructions can become confusing and mutually exclusive categories can usually be coded more succinctly using compound comparisons in the **if** object expression. (This topic is covered in the upcoming section "Compound Relational and Equality".) The following analogous construction is, however, frequently encountered:

```
If (expression1)
{ statement1a
...
    If (expression2)
    {statement2a
    ...
    }
```

If expression 1 is true, then the statements from statement 1a to the nested **if** statement (block 1) are executed regardless of the truth outcome of expression2. Statements 2a onward (block2) are dependent on this truth value of expression 1 *and* expression2 (i.e. both must be true). A construction of this type must be used rather than two unnested if's when the execution of the outer block can potentially change the truth outcome of expression 1.

Multiple if-else Statements and Indentation

Indentation should correspond to the hierarchical level of a construction except for the case of the else-if construction. Besides the syntax models, two rules govern the meaning and hierarchy of if-else as well as other control flow statements.

RULE 1 Every simple statement is controlled by the most recent if or else keyword. If two or more independent (non-nested) simple and/or compound statements appear sequentially between two control flow keywords* on their level, then these statements must be surrounded by braces to form a block. This newly-formed block is then controlled by the most recent if and else.

RULE 2 Within the same block, each else is matched to the most recent if not already matched with an else. Indentation should reflect this matching.

---

* goto is a exception.

The first rule may be restated as: "each control flow key-word except **goto** may only have one object statement, be it simple or compound. Other statements will be interpreted as independent and such statements always terminate nesting on that level." The following program section:

```
if (expression)
    statement1
    statement2
else
    statement3
```

would be illegal because statement2 would be interpreted by the compiler as independent (i.e. not part of the **if** clause). That being the case, the **else** expression would then be interpreted as beginning a new statement which, of course, is illegal. Either statement2 must be taken out of the **if-else** conditional statement altogether or a compound statement must be made of statements 1 and 2 as shown below:

```
if (expression)
    { statement1
    statement2
    }
else
    statement3
```

Although rule 2 seems obvious, troublesome errors are often made because it is violated. Consider the following section:

```
if (expression1)    /* st1 */
    if (expression2)    /* st2 */
        {statement3a
         ...
        }
else        /* st2 */
    {statement3a
     ...
    }
```

Despite what is implied by the indentation, the else expression is connected with the second if because it is the most recent. The else expression should be indented four more places as shown below to reflect the nesting hierarchy:

```
if (expression1)    /* st1 */
    if (expression2)   /* st2 */
        {statement3a
            ...
        }
    else          /* st2 */
        {statement3a
            ...
        }
```

If, on the other hand, the else section is supposed to nest with the parent if, then either inner braces may be used to confine the nested if section as in:

```
if (expression1)    /* st1 */
    { if (expression2)   /* st2 */
        {statement3a
            ...
        }
    }
else          /* st1 */
    {statement2a
        ...
    }
```

or an intervening else with a null statement may be used to "marry off" the inner if:

```
if (expression1)     /* st1 */
    if (expression2)   /* st2 */
        {statement3a
          ...
        }
      else     ;    /* st2 */
else     /* st1 */
    {statement2a
      ...
    }
```

Suppose that the unindented and largely unblocked section of code, depicted in the Initial Code Section of figure 5.3 on page 202, was presented for correction. (The abbreviations **expr** and **state** stand for expression and statement, respectively.) This section of code can be almost entirely corrected using only rules 1 and 2. The first step, as shown in figure 5.3, is to determine what statements are object statements, and with which **if** or **else** statements these are associated. Statements 1 and 2 either should be part of the block under the first **if**, or statement 2 is an independent statement. If the latter is the case, then the **if** on line 4 begins a new statement. As will be shown shortly, this second proposition is illegal.

A brace is placed before statement1 to denote a compound statement. The block end will be determined later. A brace is placed before statement 3 on line 5 for the same reason.

Notice that two **else**'s remain in this code section. These must be associated with the most recent **if**'s in the same block. The **if** on line 6 is disqualified because it is already in a block exclusive of the **else**'s. The two **else** keywords, therefore, must associate with the only two remaining **if** keywords on lines 1 and 4. This explains why this structure must (almost) entirely be a nested or chained one rather than a series of separate conditional statements.

Because the **else** on line 9 is matched to the **if** on line 4, every statement in between must be surrounded by braces to

form a compound statement. Accordingly, a brace is placed on line 8 after the original block brace. The if on line 4 has a compound object statement which is itself composed of a simple statement (state3) and another compound statement or sub-block. By the same token, a brace is placed at the end of line 10.

The first if (line 1) denotes an if-else construction that *seems* to comprise this whole section of code. Statement 10 cannot be assumed to be connected with the last else. Rule 1 does not apply to this situation since statement 10 is not situated between two control keywords (on any level). In the absence of any contrary evidence, statement 10 then must be treated as an independent statement. Braces are not positioned in place of question marks in figure 5.3.

The second step, as shown in figure 5.3, involves indenting the lines in correspondence with syntax models and subblock convention. Closing braces are placed on separate lines underneath their counterparts. Observe that statement 5 in line 8 is in line with the if in line 6. From this, one can correctly infer that statement 5 need not be associated with that if keyword. Because statement 5 is not enclosed by two keywords on its level, the construction does not violate rule 1. In the absence of any evidence to the contrary, it must be assumed that statement 5 is not object to the third if.

At this level the situation is analogous to that encountered with statement 10. The block or compound statement of which statement 5 is a member is, however, still the object of the if on line 4. Consequently if expression 2 evaluates to true, then statement 5 will always be executed regardless of the truth outcome of expression 3.

The sub-block containing lines 6 thru 9 is indented from line 5 not because it is nested to statement 3, but simply because it is a sub-block. The if statement in lines 6 and 7 and statement 5 form a block that is on the same control flow hierarchical

Initial Code Section	First Step	Second Step
```if (expr1)```	```if (expr1)```	```if (exp1)```
```state1```	```(state1```	```(state1```
```state2```	```state2```	```state2```
```if (expr2)```	```if (expr2)```	```if (exp2)```
```state3```	```(state3 /* 5 */```	```(state3```
```(if (expr3)```	```(if (expr3)```	```(if (exp3)```
```state4```	```state4```	```state4```
```state5)```	```state5))```	```state5```
```else```	```else```	```)```
```state6```	```state6) /* 10 */```	```)```
```else```	```else```	```else```
```(state7```	```?(state7```	```state6```
```state8```	```state8```	```)```
```state9)```	```state9)```	```else```
```state10```	```state10 ? /* 15 */```	```(state7```
		```state8```
		```state9```
		```)```
		```state10```

Figure 5.3.   A section of unblocked, invalid code is presented in the far left column. The middle column demonstrates proper block structure, while the rightmost column illustrates the proper associated indentation.

level as statement 3. Such sub-block groupings do differentiate the enclosed sections of code for purposes that will be discussed in chapters 6 and 7.

This text will refer to brace-delimited sections of code as compound statements when they serve as the object "statement" of control flow constructions. Otherwise brace-enclosed sections of code will be referred to as (sub-) blocks (or function body for the obligatory outermost braces.)

Compound Comparison Tests

It was stated earlier that nested simple if's are rarely used because of the convenience of compound relational and equality conditions. These compound comparisons are comprised of two or more simple relational and/or equality expressions joined with logical operators — ||, **&&,** and ! . For n comparison expressions, n-1 logical operators are required to link them to form a valid compound condition. Mutually exclusive categories can always be obtained through the proper coding of compound conditions, although overuse of compound expressions may cloud program clarity.

The example on page 196 may be written in an equivalent form using compound conditions as illustrated on page 204. The first expression specifies that the variable **grade** will be assigned a value of E if the variable **score** holds a value greater than or equal to zero *and* less than or equal to sixty. That is, a letter grade of E will be assigned to a student with a score between zero and sixty inclusive. If the score is not within these limits, the variable **grade** is not assigned a value (i.e. the object statement is not evaluated) and control passes to the next statement.

The next four **if**'s are executed in a comparable manner. the last **if** on lines 11 and 12 is unchanged from the original version because the category is exclusive as stated. Because all

of the statements in the example below contain conditions involving mutually exclusive categories, they may be arranged in any order desired.

```
if ((score >= 0) && (score <= 60))
     grade = 'E';
if ((score > 60) && (score <= 70))
     grade = 'D';
if ((score > 70) && (score <= 80))
     grade = 'C';
if ((score > 80) && (score <= 90))
     grade = 'B';
if ((score > 90) && (score <= 100))
     grade = 'A';
if (score == 666)
     grade = 'T';
```

The error condition could also have been coded using a compound condition.

```
if ((score < 0) || ((score > 100) && (score != 666)))
   grade = 'R';
```

The AND subexpression evaluates to true for all values of **score** greater than 100 except 666. This is OR'd with a subexpression which is true for negative scores. Although additional associative operators (i.e. parentheses) were placed around each simple condition, these were unnecessary because the logical AND, &&, and the logical OR, ||, operators have lower precedence than the relational or equality operators. Also the pair of parentheses surrounding the AND expression in the error checking condition are unnecessary due to the fact that AND operations take precedence over OR operations.

The logical negation operator (NOT), !, is a unary operator and as such has a higher precedence than the comparison

operators. Although single variables may be used as the operand of a logical negation operation, it is common to use this operator on an entire comparison expression. In such cases, parentheses must be used to surround the operand expression. The first **if** condition could have been coded utilizing NOT operations as follows:

If (!(score < 0) && !(score > 60))

or alternately as:

If (!((score < 0) || (score > 60)))

The original **if** condition is preferable as neither of the conditions using NOT's have as clear a meaning.

The reader must be careful to avoid syntax and logical errors when translating common English conditions into C code. The condition "if x is greater than zero or less than ten", if coded verbatim, would be:

If (x > 0 || < 10)

This construction violates C syntax rules because the "less than" relational operator has no operand preceding it. This kind of construction is provided for in some languages as it saves coding and typing time. This advantage is offset by the more complex compiler software required and longer compilation times.

Logical errors are sometimes produced by erroneous conceptualization or wording of the condition. Some common and accepted English condition formations are, in fact, logically incorrect. For example, suppose a statement is to be executed for all values except zero and ten. Many would state this condition as "if the value is not zero or the value is not ten" and accordingly would code it as:

If ((value != 0) || (value != 10))

206 C User's Handbook

Closer inspection of this **if** expression will reveal that it is true in all cases. A variable cannot equal zero and ten concurrently. The correct formation is "if the value is not zero *and* the value is not ten."

The logical AND and OR operators in C possess the useful attributes of "short circuit evaluation" and "forced order or evaluation".

These qualities allow termination of a left-to-right evaluated expression immediately after its value has been fixed. Specifically an AND string is terminated upon the first occurrence of a false value, while an OR string is terminated upon the first true value. See the logical operator section in chapter 4 for a more complete discussion.

The Switch Statement [Advanced]

C includes one other purely selectional statement other than **if-else**, the **switch** statment. It has the form:

```
switch (expression)
  { case constant expression1 :
        statement1a
        statement1b
        ...
    case constant expression2 :
        statement2a
        statement2b
        ...
    . . .
    case constant expressionN :
        statementNa
        statementNb
        ...
    default :
        statementDa
        statementDb
        ...
  }
```

where the **default** or any particular **case** section is optional. The switch's object expression can be of any sort as long as it resolves to an integral value. The object expressions of the **case** keywords must also resolve to an integral value; however, these must be constant expressions — always evaluating to the same integral value. In addition each constant expression must evaluate to a unique integral value. Duplicates are not allowed. The **case**'s and the **default** can occur in any order. Notice that, unlike **if-else**, braces need not surround each group of simple statements, although a pair are required to group the entire **case** section.

During execution of the statement, the object expression is evaluated first. The constant **case** expressions are then serially evaluated until a value is found that equals the value of the object expression. This **case** expression is then "switched on", and control is passed to the first (sub) statement in its section. All statements, not only within the activated **case** but also throughout the rest of the construction including the **default**, are then executed. This flow procedure is known as "fall-through" execution. If no object expression/**case** constant expression match is found, the statements, if any, following the **default** keyword are executed.

A more demonstrative illustration is presented by the program and several trial output runs in figure 5.4. Notice the use of symbolic code in the first two declarations of program 5.1. These symbolic strings indicate the correct C code suitable for the corresponding positions. This **flag** should either be initialized to a value of 1 or 0.

The body of program 5.1 is comprised of three declaration statements; the **switch** statements of lines 9 thru 30 and the three printf() statements on lines 8, 31, and 32. All variables are initialized in their respective declarations. These values are output, for clarity's sake, in the printf() message of line 8. The **switch** clause, which is next encountered, activates one of the **case** code sections, in accordance with the value of **trip**. Con-

trol then passes through this construction as previously described until the closing brace of the switch is encountered, at which time control passes to the statement following the switch. Note that unlike other control flow statements, multiple case object statements do not have to be enclosed in braces to form a compound statement. However, the entire case section needs to be surrounded by braces.

Consider trial run #1. Since **trip** was initialized to 5, the first case section is activated. Its only object substatement outputs the tag message: case 10. Control then passes to the second case, which begins on line 13.

This case's first object substatement on line 14 outputs a tag message. Then because **flag** was initialized to 0, the compound object statement under the else-if portion of the conditional construction of lines 15 thru 22 is executed. As a result, a tab and the string "FALSE" is output, and **flag** is reset to one.

The next case's only substatement, which occupies line 25, is then executed. (Notice that the associated constant value is an escape sequence which is equivalent to the value zero). Control passes to the case default and its two object statements are then executed. Next, line 30 is encountered and the switch statement is terminated. The program ends after the two additional printf() 's on lines 31 and 32 are performed.

The reader should likewise follow through the other trial runs presented in figure 5.4. Note that the case positions do not correspond to their constant object expression's numeric value. Also when a prior case is bypassed, its object substatements are quite naturally not executed. As this program is set up, the default case will always be executed either in a fall-through manner or singularly if none of the prior case's are switched on by **trip**.

```
/*    Program 5.1 - example switch use */

main()

{  int flag = 1;
   int trip = 10;
   short int d_fl = 0;
printf("\n flag(1or0)  trip   ");
scanf("%d   %d",&flag,&trip);
   printf("\nflag = %d, trip =%d, d_fl =%d");
   switch (trip)
      (case (5):                      /* 10 */
          printf("\ncase 5");

        case (5 + 5):
           printf("\ncase 10");
           if (flag == 1)
               printf("\tTRUE");
           else if (flag == 0)
               { printf("\tFALSE");
                 flag = 1;
               }                       /* 20 */
           else
               printf("\n *FLAG ERROR*");

        case ('\000'):
           printf("\ncase 0");

        default :
           printf("\ndefault");
           d_fl = 1;
      }                               /* 30 */
   printf("\nswitch executed");
   printf("\nflag = %d, trip =%d, d_fl =%d");
}
```

Figure 5.4. Program illustrating the use of the switch
construction

```
flag =0, trip =5, d_fl =0
case 5
case 10 FALSE
case 0
default
switch executed
flag =1, trip =5, d_fl =1
```

```
flag =1, trip =10, d_fl =0
case 10 TRUE
case 0
default
switch executed
flag =1, trip =10, d_fl =1
```

```
flag =0, trip =0, d_fl =0
case 0
default
switch executed
flag =0, trip =0, d_fl =1
```

```
flag =1, trip =15, d fl =0
default
switch executed
flag =1, trip =15, d_fl =1
```

The Switch Statement with Breaks

Often a fall-through execution approach is highly unde-sirable because execution of one or only a few case object statement sections are required. Three statements are available to obtain the case integrity being sought: the goto, if-else, and break. Use of the goto statement is discouraged on con-ceptual grounds as discussed in the section of that name, while the use of if-else statements within switch statements defeat the simplicity of the latter.

While the break statement is detailed later in this chapter, it can be succinctly defined here as follows: the break state-ment causes an exit from the control flow statement in which it is encountered. Therefore if a break statement is placed at the end of a case object statement section, then after execution of that section, control passes to the statement following the entire switch statement.

Examine program 5.2 in figure 5.5. Note again that the right side of the assignment on line 6 denotes that one of the values listed must be chosen, and output corresponding to each of these initial values is shown below the program. When **pick** equals 5, the object statements of **case 5** are executed. The printf() statement on line 9 outputs a new line followed by its case message. The break statement on line 10 instructs the computer to exit the switch. Line 21 is the only independent statement remaining in the program. It outputs two tabs and a program termination message. Control then passes to the closing brace of the function body where the program terminates.

When **pick** equals 10, **case 10** is executed; and a newline and case message are output. Since no break or other control flow command follows, control passes on to the next statement, which is located on line 14. This statement is also a case message which is output. Likewise line 16's message is output.

After line 16 a default statement appears. Control is passed outside the switch to line 22 where a printf() call outputs a "pick value" message.

The next two **pick** values of 15 and 20 operate in a similar manner except that the higher case sections are progressively skipped over until a **pick** value / case constant match is found.

The last **pick** initial value does not match any case constant. The default case is executed causing a newline, a tab, and the word "default" to be written. Here control naturally passes out of the switch construction when it encounters the closing brace of the switch. Line 22 is then executed.

Program 5.2 demonstrates fall-through and break concepts. When the output contains more than one case message (say n case messages), then one or more fall-through conditions (actually n-1) are guaranteed to have occurred. Fall-through construction is used whenever a multiple but varying number of occurrences of a single action or concatenation of action is desired (see program 5.1).

```
                    /* Program 5.2 */
/* demonstration of usage of fall-through vs.
   break control flow  in a switch statement  */
main()
                                        /* 5 */
{  int pick = 5, 10, 15, 20, or other ;
   switch (pick)
     {case 5:
         printf("\n case 5");
         break;                         /* 10 */
      case 10:
         printf("\n case 10 and");
      case 15:
         printf("\n case 15 and");
      case 20:                          /* 15 */
         printf("\n case 20");
         break;
      default:
         printf("\n default");
     }                                  /* 20 */
  printf("\t  The End ");
}
```

```
     case 5          The End
```

program output continued on next page

Figure 5.5. Example program and output illustrating the use of break's within switch statements

```
case 10 and
case 15 and
case 20          The End
```

```
case 15 and
case 20          The End
```

```
case 20          The End
```

```
default          The End
```

Figure 5.5. (cont.) Example program and output illustrating the use of break's within switch statements.

The **break** construction is used when unrelated or conflicting actions are to be executed within the **case**'s or when program modification is common. For both types of **switch** construction, it is a good programming practice to include a **break** statement as the last line of the last **switch**. This protects against natural fall-through to any **case**'s that might be added later.

C relies on the programmer to add **break** statements. Automatic break mechanisms were not built into the **switch** statement for the following reasons:

Simplicity	Inclusion of such a mechanism would result in a slightly more complex compiler and slower compilation.
Flexibility	An automatic break construction eliminates fall-through options, thereby reducing the generality of use.

This is another example where C's simplicity translates into a strength.

The Conditional Expression vs. the if-else Statement

In chapter 4, C's only ternary operator, the conditional operator, **?:**, was presented. Many beginning programmers are confused by the difference between conditional expressions and **if-else** statements. While both provide a control flow capacity, there are important differences in syntax requirements and usage.

A brief explanation of the conditional operation and the **if-else** statement follows:

expresssion1 **?** *expression2* **:** *expression3*

(Sub)expression1 is evaluated first. If it is true (non-zero), then (sub)expression2 is evaluated and expression3 is not evaluated. The resultant value is that returned by the entire condi-

tional expression. If expression1 evaluates to false (zero), expression3 is evaluated, while expression2 is not. The value of expression3 is then the value returned by the entire conditional expression.

The **if-else** statement is used as follows:

```
if (expression)
    {statement1a
     statement1b
     . . .
    }
else
    {statement2a
     statement2b
     . . .
    }
```

If the object expression evaluates to true (non-zero), then the statements enclosed in the block between the **if** and **else** keywords are executed. The **else** section of the construction is ignored. If the expression evaluates to true (zero), then the **else**'s object block is executed and the first compound statement is ignored.

As demonstrated by the attendant explanations, both code groups have the same basic control flow shape. Their main differences lie in their syntax and what class of code they represent, which restricts their use.

The syntax for both requires an expression for the condition. The **if-else** statement requires object statement(s) to hinge on this evaluation while the conditional expression requires subexpressions. Although C includes a great deal of leeway as to the construction of expressions (i.e. any number of primary expressions connected by appropriate operators, including assignment operators), still expressions cannot contain control flow devices other than the ternary operator nor can they contain declarations of any type. The conditional

expression is also limited in that it can only contain one expression in each of the true or false positions (i.e. expressions 2 and 3), while the if-else statement can have a virtually unlimited number of object statements as long as they are enclosed in braces to form a compound statement.*

The second major difference may be stated as follows: *the ternary operation defines an expression while the if-else construction defines a statement.* There are specific instances where an expression must be used: as object conditions for control flow constructions, as operands for the majority of C's operators, etc. In these cases, any control flow statement would violate syntax rules as in the following:

```
If ( If (a > b) a = b;)        /* Illegal*/
    n = c + a;
```

Not suprisingly, the if-else construction can only be used in those instances that demand a statement. However the conditional operation can qualify for these requirements by simply suffixing the expression with a semicolon, thus forming a valid statement. (Remember from chapter 4 that any expression can likewise be transformed into a statement.) Hence the conditional expression is more flexible than the if-else statement in that it has a much wider area of application but is less flexible in that it will only accept one expression per true or false position.

The scanf( ) Input Function

The scanf( ) function is the input complement to the printf( ) output function. Both of these functions are often used in applications programs. Because of their ability to han-

---

* Actually any number of independent subexpressions, connected by the sequence (comma) operator, can be used as subexpressions in the ternary operation. There are significant restrictions on this type of "expression concatenation".

dle a field of characters in a selective manner, these two routines are sometimes referred to as "string" or "formatted" I/O routines. (There are two corresponding character I/O routines, getchar( ) and putchar( ), that are discussed in chapter 11.)

The scanf( ) function call looks much like a printf( ) call except the former may not contain escape sequences and typically makes use of the "address of" operator, &, to prefix variable identifiers. A scanf( ) function call takes the following form:

scanf(*control string, exp1, exp2, ..., expN***);**

where exp1, exp2, through expN are expressions resolvable to a valid *pointer value*. A pointer value is the lvalue of a previously declared variable. For the fundamental data types discussed so far, a corresponding pointer value can be obtained by prefixing the variable identifier with the unary "address of" operator, &. Although the full significance of this operation will be explained in chapter 9, let it suffice for the present to state that a pointer value (lvalue) is needed to guide the inputted value (rvalve) to the correct block in memory.

The control or conversion string for scanf( ) is similar to its printf( ) equivalent in that it can contain:

- Whitespace (i.e. tabs, blanks, and newlines) which are ignored.
- Conversion specifications, which are comprised of the percent character, followed by an optional *assignment suppression character*, *, then a conversion character.
- Other characters, beside whitespace and conversion specifications, which are interpreted as *matching characters*.

Conversion specifications and literals in the control string manage the division of the *input stream* into *input fields*. The input stream is simply the string of all characters input to the

computer (here during the scanf( ) process). An input field is a section of the input stream that is uniquely identified as a unit by the control string. The following conversion characters correspond exactly to the ones presented for printf( ):

integer flavored input:

d	A decimal integer is expeceted in the input stream.
o	An octal integer is expected in the input stream. The object numeric string may optionally begin with a zero.
x	A hexadecimal integer is expected in the input stream. The object numeric string may optionally contain a leading **0X** or **0x**.
h	A short integer is expected in the input stream.
c	A single character is expected in the input stream. The normal skipping of whitespace characters does not occur.

floating point flavored input:

f,e	A floating point number is expected in the input stream. It may either be in simple decimal or exponential format and either conversion character may be used as the two are synonymous.

character string input:

s	A non-white character string is expected in the input stream. The corresponding argument in scanf() list should be a pointer to a string of appropriate size. The terminating null character will automatically be appended by scanf(). Strings will be discussed in chapter 9.

The conversion characters **d**, **o**, and **x** can be prefixed with a lowercase "ell" (l) to indicate that the input string field is

expected to represent a **long int** data type of the corresponding base. On some compilers this **long** designation can be implemented by the use of capitalized conversion characters — **D** in place of **ld**, etc.

Normally each identified input field is assigned in a sequential manner to the variables indicated by the pointer values. Assignment can be suppressed by preceding a conversion character with an asterisk, *. When this suppression character is properly included within a conversion specification, the input field defined by this specification is skipped over; the normal assignment is not made. Consider the following code section:

```
int k, l;
float w;
double x = .111e-6;
scanf("%d %o %e %*c %le",
         &k, &l, &w, &x);
```

and the unencoded input stream:

30   30   30.12345678 Z 30.12345678

Variable **k** is assigned a value of decimal 30 and l a value of octal 30 (decimal 24). On the reference system, **w** was assigned a value of decimal 30.123455 and x a value of 30.1234567. The assignment of the character Z was suppressed by the %*c specification.

Input code must be separated in some manner so that scanf( ) can unambiguously divide the input stream into data fields. Consider the program and its output, presented in figure 5.6. Study how the conversion specification in the scanf() of line 13 divides the user supplied input stream. (The underlined code represents input supplied by the user through the "stand-

ard input"). The following points are exemplified by the example program run:

- Whitespace in the control string itself is ignored.

- Whitespace in the input stream is usually ignored except for the purpose of delimiting data fields. The exception to this rule occurs with the **%c** specification; then whitespace is considered valid data. Hence variable c1 is assigned a value of a blank, ASCII 32.

- The suppression specification **%*s** caused the character string **elbronS1#** to be skipped. Non-white-space characers will not automatically be skipped by scanf( ) in order to find an appropriate field.

- The matching characters **: S T O P** demand the inclusion of the specified characters in the proper place in the input stream. Only white characters will be skipped over to find matching characters.

- In the absence of delimiting whitespace, scanf( ) utilizes invalid characters (relative to the current specification) to determine the end of an input field. Thus the run-on section of the given input line was divided after the matching colon character, after another character (**q**), after each matching character in the group **STOP** (there could have been intervering whitespace between these characters), and before the decimal point since this is not a valid integer character.

The scanf( ) will not perform data type conversions although truncation of "insignificant" digits will occur as expected. Failure to correctly convert a field often results in automatic conversion errors in subsequent conversions within the same scanf( ) call. This routine does not provide for error handling, although it does "send back" sufficient information to police the input process (as will be shown shortly).

```
            /* Program 5.3 */
   /* example of scanf() routine */
main()

{   char c1, c2, c3;
    int i1, i2, i3;
    long l1;
    float f1;          /* 8 */
    printf("\nInput the following quan");
    printf("tities: \nint string int ");
    printf("char :char STOP long float\n");
                        /* 12 */
    scanf("%d %*s %d %c : %c STOP %ld %f",
            &i1, &i2, &c1, &c2, &l1, &f1);

    printf("\n i1 = %d, i2 = %d,", i1, i2);
    printf(" c1 = %c, c2 = %c,", c1, c2);
    printf("\n l1 = %ld, f1 = %f,", l1, f1);
}                       /* 19 */
```

```
Input the following quantities:
int string int char :char STOP long float
111 eibronS1# 22    :qSTOP12345678.43434e3

i1 = 111, i2 = 22, c1 =  , c2 = q,
l1 = 12345678, f1 = 434.339996,
```

* Where underlined characters represent input from user.

Figure 5.6. Example of the use of the scanf library routine.

Reference Program: The First Version

By now, the beginning C programmer should have an understanding of all the elements of C necessary to code a solution program to the reference problem: data types, operators and syntax, one-function program format, I/O routines, punctuation, and the conditional **if-else** statement. A typical solution program is depicted in figure 5.7. The body starts with three declaration statements. Since it was never stated as a premise in the reference problem that scores are only to be integral, the variable **score** must be declared as **float**. Line 8's **printf()** call outputs a blank line followed by a prompt message to the standard output device. It ends by returning the cursor or carriage control so that data entry can start on a new line.

The **scanf()** function call on line 9 simply assigns the input values returned by the standard input device to the variables pointed to in the argument list. The first input value must be a continuous numeric string which is treated as a long decimal integer and assigned to **st_num**. A comma must be the next non-whitespace character. It is followed by an arbitrary number of whitespace characters and another numeric string. This second numeric string is interpreted as a floating point decimal value which is subsequently assigned to **score**.

The 10th line starts a nested **if-else** statement (utilizing the **else-if** construction variation) that ends on line 23. Exclusive category conditions and simple assignment object statements are used at each level. **printf()** calls on lines 24 and 25 output labels and the corresponding student/grade information.

Several test runs are included below program 5.4 in figure 5.7. Test data should elicit different responses from a program and involve end conditions of deeply nested structures. The test data presented in figure 5.7 involves three different conditions and corresponding output. Also the error condition score of 235 checks control flow to the last executable object statement, located on line 23.

```
/*      Program 5.4 - First solution program to
    this chapter's reference problem      */
main()
{   char grade;                    /* 5 */
    long int st_num;
    float score;
    printf("\nEnter student number,      test score. \n");
    scanf("%ld , %f",&st_num,&score);
    if ((score >= 0) && (score <= 60))        /* 10 */
        grade = 'E';
    else if ((score > 60) && (score <= 70))
        grade = 'D';
    else if ((score > 70) && (score <= 80))
        grade = 'C';                   /* 15 */
    else if ((score > 80) && (score <= 90))
        grade = 'B';
    else if ((score > 90) && (score <= 100))
        grade = 'A';                   /* 20 */
    else if (score == 666)
        grade = 'T';
    else
        grade = 'R';
    printf("\n\n Student number- %ld      Score- %f ",st_num,score);
    printf(" Grade-%c\n",grade);          /* 25 */
}
```

```
Enter student number,   test score.
123456789  ,  79  .

       Student number- 123456789    Score- 79.000000    Grade-C


Enter student number,   test score.
526219488  ,  666

       Student number- 526219488    Score- 666.000000    Grade-T


Enter student number,   test score.
293467070, 235

       Student number- 293467070    Score- 235.000000    Grade-R
```

Figure 5.7. The first version of the program designed to handle this chapter's reference problem.

PROGRAM ENHANCEMENTS

Although program 5.4 is adequate, it could be improved in several different ways. First, because the scanf() function does not ordinarily output error messages, it is often useful to include a short output message that signals input acceptance. This can be a very useful tool both as a message to the user indicating that everything is preceding normally and as a debugging tool to indicate that possible *lock-outs* did not occur during statement execution. Also it is often desirable to highlight error conditions with an additional descriptive message. The program can even be designed to optionally end when such an error condition occurs.

A second possible enhancement deals with *user-friendliness*. A user-friendly program is one that is easy to use. A user-friendly program explains input requirements, output, and provides for gracious error handling, including descriptive error (and warning) messages. What may appear as an obvious and friendly program to the programmer may seem baffling and frustrating to the user.

The last two suggested improvements also deal with the adequacy, from both a form and content perspective, of the output. The output should be well-organized and should present the required information in a logical manner.

Now examine program 5.5 as presented in figure 5.8. The first improvement injected into the new version is the inclusion of the variable **test**. A prompt for this variable is output via line 9 and an input integer value is assigned to it in the next line. Eventually this piece of information will be output as a header (see trial runs). This provides a more detailed output (as this program could now be used over several tests without confusing output data). Since the header is centered over the two output labels it concerns, it is conceptually pleasing.

The second change occurs on line 11. The string **(comma required)** has been added to the printf() call. Although inclu-

sion of this phrase may seem like a trival point, without such explicit remarks it is not unusual for uninitiated users to encounter input format errors. Explicit input documentation is an essential feature of a user-friendly program.

Line 14 is also new. This statement simply outputs an input data acceptance message. This is generally more useful in larger interactive programs where no immediate output is forthcoming, howver inclusions of this kind are not a detriment to program operation.*

The first two printf() prompt / scanf() input pairs do not require such corresponding acceptance messages because another prompt occurs shortly after their execution. In this way the user knows that scanf() has accepted values, although he has no way at that time of knowing if these values were converted and assigned correctly. A somewhat thorough but simple input-checking procedure will be discussed later in this chapter.

The if statement added on lines 33 and 34 outputs an additional message when an error condition occurs, as demonstrated by the second trial run. This is a very simplistic error handling procedure. Often error handling involves calling entire functions and may involve program reset or abort routines.

The last improvement is the format specifications added to line 30. They specify that **st_num** is to output in a field width of 9 while **score** is to occupy a field width of 5 and have two fractional digits or "digits of precision". Formatted output in the statement serves both to suppress insignificant digits in the output of **score**'s value and to retain proper alignment with respect to other output lines. Without the field width specification on **st_num** (line 30), most of the last output line in run #2

---

* Note that this acceptance message is printed below the input line although no newline escape sequence has been previously supplied. Standard scanf() routines will automatically reset cursor or carriage control after a printf() call.

```
            /* Program 5.5 */
   /* Slightly enhanced version of Program 5.4 */
main()

{  char grade;          /* 5 */
   int test;
   long int st_num;
   float score;
   printf("\nEnter test number - ");
   scanf("%d",&test);                       /* 10 */
   printf("\nEnter student number, (comma required)   ");
   printf(" test score \n");
   scanf("%ld , %f",&st_num,&score);
   printf(":OK\n");
   if ((score >= 0) && (score <= 60))        /* 15 */
      grade = 'E';
   else if ((score > 60) && (score <= 70))
      grade = 'D';
   else if ((score > 70) && (score <= 80))
      grade = 'C';                           /* 20 */
   else if ((score > 80) && (score <= 90))
      grade = 'B';
   else if ((score > 90) && (score <= 100))
      grade = 'A';
   else if (score == 666)                    /* 25 */
      grade = 'T';
   else
      grade = 'R';
   printf("\n\t\t\t\t\tTest #%d",test);       /* 29 */
   printf("\n  Student number- %9ld    Score- %5.2f ",
                              st_num, score);
   printf("   Grade-%c",grade);
   if (grade == 'R')
      printf("\n\t\t\t\t    **INPUT ERROR**");
   printf("\n");                             /* 35 */
}
```

Figure 5.8. Enhanced version of the program presented

program output on next page

```
Enter test number - 1

Enter student number, (comma required)   test score
287680186 , 99.57
:OK
                                Test #1
      Student number- 287680186     Score- 99.57    Grade-A
```

```
Enter test number - 1

Enter student number, (comma required)   test score
00023023 , 101.46
:OK
                                Test #1
      Student number-     23023     Score- 101.46    Grade-R
                                **INPUT ERROR**
```

Figure 5.8. (cont.) Enhanced version of the program presented

would be shifted to the left seven places. This is due to the fact that the value of **st_num** in this example has seven fewer digits than the values used for this variable in previous runs.

Several serious flaws remain associated with program 5.5. It is still somewhat unfriendly. For example, the program will still accept garbage input values. Also note that tests are assumed to be 100 points in maximum value. This compromises both friendliness, as the user is not notified of this assumption, and generality, as it severly limits the utility of the program. Perhaps program 5.5's biggest shortcoming lies in its inability to accept more than one set of input values per run. The second half of this chapter deals mainly with constructions in C that handle loop or iterative control flow. These constructions allow repeated procedures, as called for in this problem's initial conceptualization.

The "Brute Force" vs. the Iterative Approach

There are two methods which may be used for altering program 5.5 (figure 5.8) so that it will accept ten sets of input values — the "brute force" and the iterative or looping constructions. The brute force approach entails recoding any section of code as required. For ten input values, lines 11 through 34 would have to be repeated ten times (assuming all the scores are associated with the same test number). The left-hand diagram in figure 5.9 shows the flowchart for this approach.

The second approach uses a loop to repeat the section of code labeled "main routine." (i.e. lines 11 through 34, program 5.5). As the diagram in figure 5.9 demonstrates, a loop has the desirable attribute of repeating a section of code without physically duplicating this section. The illustrated loop will terminate after the loop executes ten repetitions (i.e. loop repetitions $>= 10$).
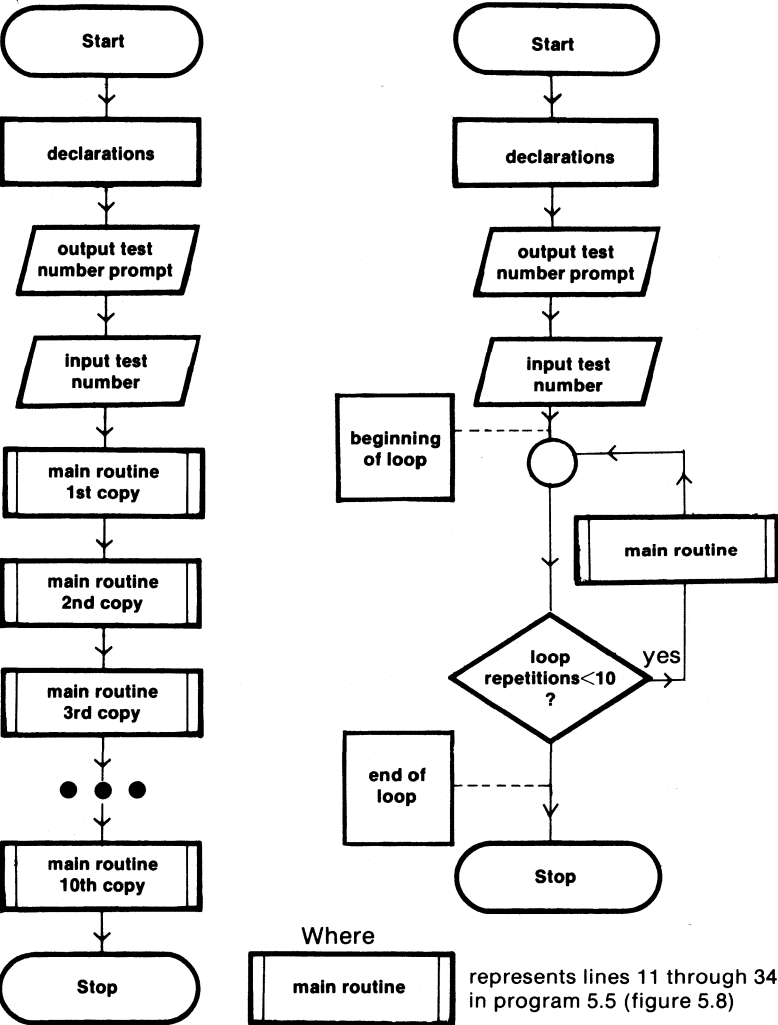
Figure 5.9. Flow diagram for the "brute force" and loop constructions necessary to convert Program 5.5 so that it handles ten input values.

There are two obvious disadvantages of the brute force approach when compared to the iterative. First, without an editor capable of copying groups of lines, typing in this program would take considerably more time. Secondly, both the source and object code would require greater storage capacity.*

Just as important, however, is the inherent flexibility afforded by loop constructions. This flexibility results mainly from the ability to alter the condition that controls loop repetition. This is accomplished by means analogous to those used to control the if-else conditional statement.

Introduction to Loops in C

As demonstrated in figure 5.9, looping constructions are an efficient means of handling a series of statements or processes (usually function calls) that must be repeated a variable number of times. Note that in the aforementioned figure, a decision and an included condition control the execution of the loop. All of the commonly used loop statements — **while**, **do-while**, and **for** — include this built-in decision step. Only the seldom-used **goto** control flow statement does not have such a step. These decision steps, often called execution (test) conditions, are generally necessary as loop termination usually occurs when this condition evaluates to false (zero). Therefore the object statements are executed over and over until the test condition is not satisfied (i.e. evaluates to false). When this occurs, loop execution ends (i.e. the substatements are not evaluated again). Control then passes to the statement immediately following the entire loop.

---

* On the author's reference system, the brute force program compared to the iterative version as follows: source and object code for the brute force approach was almost eight times as long as for the iterative approach, while the executable code length was only 1.3 times as large. This last comparison would have been more remarkable on operating systems highly compatible to the C language, such as UNIX and related systems.

Loop termination may also occur in a non-standard fashion through the use of the **break, goto** (this chapter), or **return** conditional statements (chapter 6). To terminate an otherwise infinite loop, either one of these aforementioned statements must be used, or program execution must be initiated at a higher command level, as through a program abort or a power failure.

Loop control can be broadly divided into two classes: "classical" and "non-classical" control loops. Classical loops include three main elements that control the number of repetitions they perform.

- The index or counter is a variable that counts the number of repetitions of a loop. In the most straight-forward usage, the index is initialized to zero and is incremented by one with every repetition. In general, however, the counter can be initialized to any valid value, and any *constant* value can be added to it with each pass.
- The test condition, which has the following form:

  index   comparison optr   constant expr

  is set up to terminate execution when the index reaches a certain value. In other words, this test terminates loop execution after a predetermined number of executions.
- The increment is the constant, usually integral value that is added to the index after each loop execution.
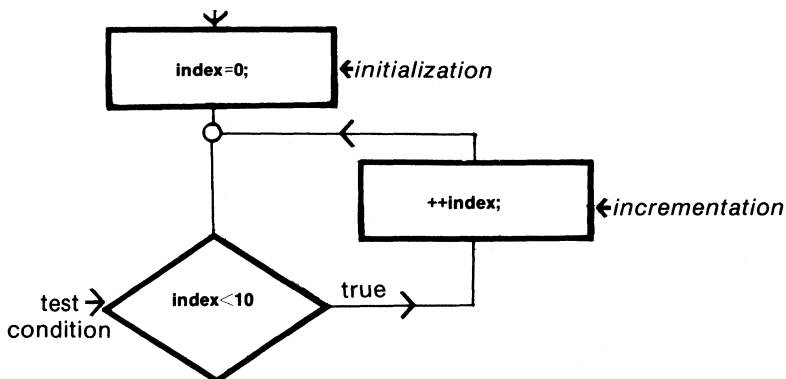
For example the following diagram illustrates a classically controlled loop. The loop executes a total of ten times. At that time, the index has a value of eleven resulting in the test condition failing and the loop being terminated.

Note that the initialization of the index occurs prior to the loop. If this step occurred in the loop itself, **index** would be reassigned a value of zero with every loop pass, and the condition would never fail. Classically controlled loops observe the following equation:

$$\text{\# of repetitions} = \frac{\text{final index value - initial index value}}{\text{increment}}$$

where the final index value is defined in the test condition.*

Non-classically controlled loops are obviously those that do not adhere to the above strict guidelines. Some programming languages have control flow statements which are structured to operate only in a classical manner. C's control flow statements are general and flexible. Like the if conditional statement, C's control flow test conditions can be any resolvable expression, while any valid resolvable statement may be used as an object statement. Also the object statement can again either be a simple statement or a compound statement —a group of one or more simple statements enclosed in a pair of braces.

The while Statement

The **while** statement's syntax is:

```
while (expression)
    { statement1
      statement2
      ...
      statementN
    }
```

---

* C's do-while loop follows this equation, except its minimum number of executions is one.

As in all other control flow statements, except **goto**, the parentheses surrounding the condition expression are mandatory.

The flow diagram for the **while** statement can be represented as:
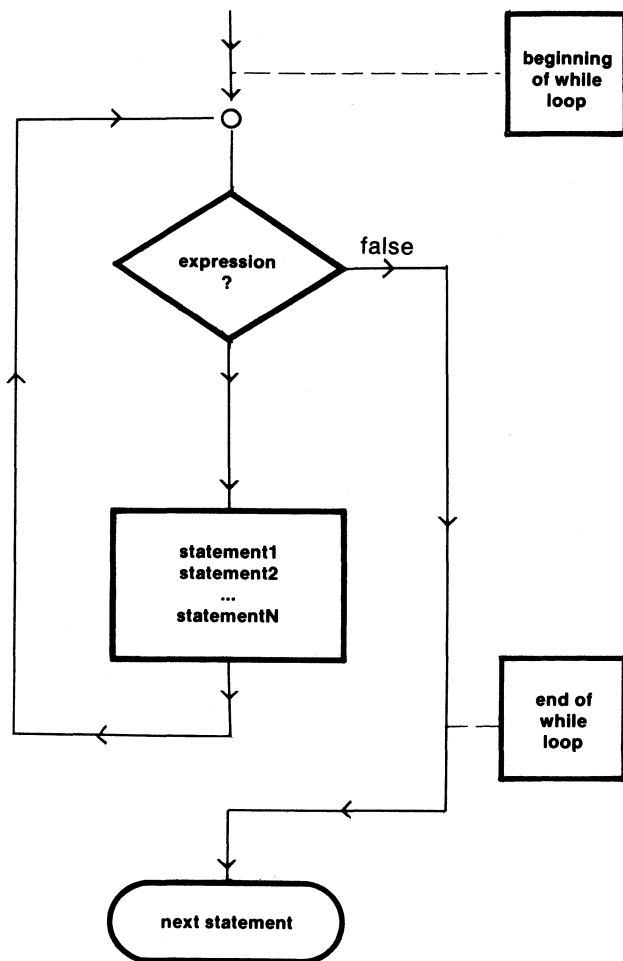
where the string "next statement" is not part of the while but represents loop termination and passage to the next statement.

The while statement flow diagram is also commonly represented as shown in the right-hand diagram of figure 5.9 If the condition expression originally evaluates to false (non-zero), then the compound object statement (or block) is ignored and control passes to the next statement. If the expression originally evaluates to true (non-zero), then the compound object statement is executed. Control then passes back to the (test) condition expression. This expression is subsequently reevaluated and the process starts over.

For normal loop termination to occur once the test has been initially evaluated to true, one or more of the statements comprising the compound object statement must affect at least one of the variables within the condition expression.

For classical loop control, the for loop statement is more commonly used. The while statement is usually reserved for non-classical loops, as exemplified by program 5.6 shown in figure 5.10. This program sums integer input quantities less than or equal to 50. The program terminates when an input value is out-of-bounds (i.e. 51 or greater) or after the sum total exceeds the limits of 0 and 100, as defined in the while condition on line 7. The parentheses within the condition on line 7 are optional, but they greatly improve readability.

The while's compound object statement occupies lines 8 through 15. An input prompt is output via line 8. Note the use of the variable count. It is used to tally the number of input values and to modify the prompt message. The next line is a simple integer scanf() assignment to variable r.

A simple construction follows that adds the input value r to total, increments the input counter, and stores the current value of r in a variable named r_old only if r's value does not exceed 50. The repetition of the initial loop condition is required because the three statements in the if are executed

between the time the scanf() assignment is made and the loop condition is checked.

Because loop termination for a out-of-bounds input value cannot occur before the end of this block, input data validation must occur within the loop (line 10). Lines 11 through 13 could have been moved before input assignment. This would, however, introduce the complication that loop termination would occur before the last acceptable input value could be added to **total**.

The last three lines of the program output summarize information. The utility of **r_old** is revealed in line 17. **r_old** is only reassigned a value in line 13 upon input of an acceptable value. Variable **r**'s value changes during scanf() execution despite the nature of the input. As its name suggests, **r_old** contains the last accepted input value. Note also that the argument in the printf() statement on line 18 is not an identifier or a constant, but the following expression:

total/(float)count

The cast operator is required to force one of the integer operands of the division into floating point representation. Otherwise, truncation of the fractional part of the quotient would occur.

Three trial runs are illustrated below program 5.6, each representing termination by a different loop criterion. Trial run #1 terminates after **total** exceeds 100, while run 2 terminates after variable **total** becomes negative. Run 3 terminates when an input value exceeding 50 is input.

Error messages, explanatory prompts, and other user-friendly devices were not included in this program in order to avoid obscuring the main points of concern. Devices of this

```
/* Program 5.6 */
/* while loop statement without loop counting criteria */
main()
{   int r=0, r_old=0, total=0, count=0;          /* 5 */
    printf("\n\n\n");
    while ((r < 51) && ((total > -1) && (total < 101)))
    {   printf("Enter integer input #%d: \t",count+1);
        scanf("%d",&r);
        if (r < 51)                              /* 10 */
        {   total +=r;
            count++ ;
            r_old = r;
        }
    }                                            /* 15 */
    printf("\n\nsum = %d,     number of inputs = %d",total,count);
    printf("\n\nlast accepted input value = %d",r_old);
    printf("\n\naverage input value = %.3f\n", total/(float)count);
}
```

Figure 5.10. Program demonstrating use of non-classical while loop.

program output on next page

```
Enter integer input #1:          34
Enter integer input #2:          -11
Enter integer input #3:          37
Enter integer input #4:          -20
Enter integer input #5:          33
Enter integer input #6:          19
Enter integer input #7:          12


sum = 104,   number of inputs = 7
last accepted input value = 12
average input value = 14.857
```

```
Enter integer input #1:          4
Enter integer input #2:          34
Enter integer input #3:          21
Enter integer input #4:          -69


sum = -10,   number of inputs = 4
last accepted input value = -69
average input value = -2.500
```

```
Enter integer input #1:          14
Enter integer input #2:          7
Enter integer input #3:          63


sum = 21,   number of inputs = 2
last accepted input value = 7
average input value = 10.500
```

Figure 5.10. (cont.)   Program demonstrating use of non-classical while loop.

kind will only be included in programs designed to illustrate practical applications rather than C language concepts.

The for Statement

The **for** statement

```
for (expression1; expression2; expression3)
    { statement1
      statement2
      ...
      statement3
    }
```

appears syntactically more complex than the while statement because of the inclusion of three expressions within the parentheses. Actually the for statement is exactly equivalent to the following two statements:

```
expression1;
while (expression2)
    { statement1
      statement2

      ...

      expression3;
    }
```

where expressions 1 and 3 have been converted into simple statements by the addition of terminating semicolons. Therefore expression1 is executed only once at the beginning of the for loop. expression2 is then evaluated. If it is false, control passes to the next sentence. If expression2 evaluates to true, then the compound or simple substatement, along with expression3 will be executed. Afterwards control is passed to the beginning of the loop where expression2 is again evaluated. The process (minus expression1) is then repeated.

The **for** statement is exceptionally well suited to handle the classically controlled loop. Consequently the loop control line for a classically controlled **for** loop would have the following syntax:

for (*index = initializer***; ***test condition***; ***index += increment***)**

Generally no requirements are associated with these expressions beyond those normally associated with other expressions (e.g. syntax, previously declared identifiers, ability to be resolved, etc.). In fact, the null expression can be used in place of these expressions. If expressions 1 and 3 are null, then **for** becomes directly equivalent to a **while** statement. If expression2 is null as in the following:

for (scanf("%d",&r); ; r++)

then expression2 is taken to be permanently true, thus defining an infinite loop. When all the required manipulation of the loop is accomplished within parentheses, the object statement should be the null statement.

Examine program 5.7 as presented in figure 5.11. The program accepts a dollar (and cents) amount from standard input. If this quantity is less than a million, it is output with the leading zeroes replaced by stars (asterisks). Such a printing scheme is commonly employed in mechanized check processing to discourage tampering.

Input prompt and assignment occurs on lines 7 and 8 respectively. Lines 10 and 11 contain the first **for** statement. The variable **w** is initially assigned the value of the input. As long as **w**'s value is larger than one, **w** is divided by ten and the

variable **hdigits** is incremented by one. Therefore **hdigits** counts the number of loop repetitions and the number of times **w** can be divided by 10 while still leaving high order (non-functional) digits. Consequently this integer indicates the number of non-fractional digits in the original input value.

The next for loop is a classical one. It simply outputs a star for every leading place out of the maximum of six not counted by **hdigits** as being occupied by a significant (non-zero) digit. Line 12 can be translated as "For variable **outstar** equals zero until **outstar** equals (six minus the number of high order digits), output a star, then add one to **outstar**."

```
                  /* Program 5.7 */
/* Outputs dollar amount with leading stars inserted */
main()

{   int digits=0, outstar;                   /* 5 */
    float indol, w;
    printf("\nInput dollar amount < 1 million: ");
    scanf("%f",&indol);
    printf("\n");
    for (w = indol; w > 1.0; w /= 10.0)       /* 10 */
        digits++ ;
    for (outstar = 0; outstar != (6 - digits); outstar++)
        printf("*");
    printf("%-3.2f\n",indol);
}                                             /* 15 */
```

Figure 5.11. A program to output an asterisk edited dollar amount.

program output on next page

```
Input dollar amount < 1 million:   34.99

****34.99
```

```
Input dollar amount < 1 million: 123456.78

123456.77
```

```
Input dollar amount < 1 million: 0000.67

******0.67
```

Figure 5.11. (cont.)   A program to output an asterisk edited
dollar amount.

Line 14 outputs the original input value. The negative sign
in the conversion string calls for *left justification*. Normally
numeric data is right justified — all the unused space allocated
for the number in the output is filled with blanks and placed to
the left of the digits. Automatic right justification would place
left spaces between the stars and digits on trial runs 1 and 3.

The do-while Statement

do-while statements are constructed in accordance with
the following syntax model:

> **do**
>   { *statement1*
>   *statement2*
>   ...
>   *statementN*
>   }
> **while** (*expression*);

where the braces are optional if the object is one simple statement. The do-while loop resembles the while loop in both syntax and operation. The main difference between these is that the execution condition occurs after the object statement in the do-while and is only checked after this substatement has been executed. Therefore, *the do-while object statement is always executed at least once*, regardless of the outcome of the condition.

Normally, iterative constructions can be more logically coded using loops with initial condition checking (i.e. the for and while statements). Nevertheless in those small percentage of instances when concluding condition checking is highly desirable, the do-while statement is a valuable tool.

Consider the operation of inputting a character value. Unlike its numeric peers, the character conversion specification, %c, does not skip whitespace. Therefore in many applications, the unconditional use of scanf() to input a character will lead to errors if leading whitespace characters are erroneously inserted. What is needed under these circumstances is a method of testing for and skipping whitespace characters.

Examine program 5.8, presented in figure 5.12. The do-while construction located on lines 9 through 11 handles the problem of locating the first non-white character and assigning it to in_chr. The condition on line 11 stipulates that scanf() should search the input line character by character until a non-blank, non-tab, non-newline character is encountered. When such a non-white character is finally assigned to in_chr, the do-while loop ends.

The if statement on lines 13 and 14 converts lowercase ASCII alphabetic characters to uppercase. The lowercase letters have a value between 97 and 122 inclusive, while the corresponding uppercase character's values are 32 less. Accordingly line 13 selects all lowercase character values, and line 14 subtracts 32 from these values changing them to upper-

```
/*          Program 5.8 - Modified single char-
    acter "echo" function that inputs a non-
    white ASCII char and outputs the uppercase
    (alphabetic) char, otherwise the same char */
main()                          /* 5 */

{ char in_char;
  printf("\nInput a character- ");
  do
      scanf("%c",&inchar);    /* 10 */
  while ((in_char == ' ') || (in_char == '\t')
          || (in_char == '\n'));
  if ((in_char > 96) && (in_char < 123))
      in_char -= 32;
  printf("%c",in_char);       /* 15 */
}
```

```
Input a character- T
T
```

```
Input a character- z
Z
```

```
Input a character- *7
*
```

```
Input a character- 198
1
```

Figure 5.12. A modified echo program demonstrating a proper application of the do-while statement.

case values. Line 15 outputs the value of **in_chr** as a character on the line immediately preceding the input line.

The **do-while** statement is advantageous in this program because at least one character value must be read, and until it is, a condition test cannot be performed. In most instances, the condition variables will already have assigned values and a **do-while** construction is not needed. The **while** and **for** loops should normally be used rather than **do-while** loops because **do-while** loops tend to be more confusing than the other loops and are consequently more prone to errors.

The loop consisting of lines 9 through 11 could be written utilizing a **while** loop:

```
scanf("%c",&in_char);
while ((in_char == ' ') || (in_char == '\t')
        || (in_char == '\n'))
    scanf("%c",&in_char);
```

When one or more of the variables in the test condition have no initial value, then those statements that are responsible for assigning these values must be rewritten before the **for** and **while** loops are encountered.

In this case, the **scanf()** statement must precede the loop in order to initialize **in_chr** so that the condition can be tested. Repetition of code in this manner can obviously be inefficient. Programmers sometimes avoid this repetition of code by assigning initial values to condition variables that will trip the loop. For example the previous **while** loop can be recoded as:

```
in_char = ' ';
while ((in_char == ' ') || (in_char == '\t')
        || (in_char == '\n'))
    scanf("%c",&in_char);
```

This practice is generally considered a poor programming practice. It is error prone and confusing because it does not convey purpose, and it is not always applicable.

The moral is: "while and for loops should be used in preference to the do-while loop construction unless at least one repetition is desired, as when initial values for condition variables must be calculated."

NUMBER OF LOOP REPETITIONS, while vs. do-while

Despite initial expectations, the number of repetitions of similar while and do-while loops is the same except for those instances when the condition is initially not met. In these cases, the while's object statement is not executed, and the do-while's object statement is executed once.

Equivalent, classically-controlled, while and do-while loops are presented in programs in figure 5.13. Both programs prompt for and input condition test values. Then for each loop repetition, a short output message containing the value of the loop counter is output. Note that for test values equal to or greater than one, the number of loop repetitions is equal. *Only for those test values that result in an initial condition evaluation of false do the number of loop repetitions differ.* In these instances, the do-while loop executes once while control passes immediately over the while's object statement.

```
          /* Program 5.9 */
     /* classically controlled
          do-while loop   */
     main()

     {   int count=0, test;  .
         printf("\nEnter loop ");
         printf("test value- ");
         scanf("%d",&test);
         do
              { ++count;
                printf("rep#%d   "
                            ,count);
              }
         while (count < test);
     }
```

```
     Enter loop test value- 4
     rep#1   rep#2   rep#3   rep#4
```

```
     Enter loop test value- 1
     rep#1
```

```
     Enter loop test value- 0
     rep#1
```

Figure 5.13. Comparison of loop repetitions for analogous while and do-while loops.

```
         /* Program 5.10 */

/* classically controlled
         while loop    */
main()

{  int count=0, test;
   printf("\nEnter loop ");
   printf("test value- ");
   scanf("%d",&test);
   while (count < test)
        { ++count;
          printf("rep#%d   "
                     ,count);
        }
}
```

```
Enter loop test value- 4
rep#1  rep#2  rep#3  rep#4
```

```
Enter loop test value- 1
rep#1
```

```
Enter loop test value- 0
```

Figure 5.13. (cont.)   Comparison of loop repetitions for
analogous while and do-while loops.

Flexibility of Loops in C [Advanced]

It was previously noted that C's control flow structures are quite ordinary except for their concise syntax and the lack of restrictions on their use. The greater flexibility of C's control flow statements can be demonstrated in three primary areas:

- Control may pass into or out of any one of the simple statements comprising the object statement of a conditional or loop via a label/goto statement combination.
- Any type of object expression(s) or object statement may be used as long as it meets normal criteria for such code.
- Variables associated with the object expression(s) ("loop variables") can have their values changed by occurrences within the object statement.

The first point can be demonstrated by the following symbolic program section:

```
. . .
goto label1;
. . .
while (expression)
    (statement1
    statement2
        . . .
    label1: statement
        . . .
        if (expression)
            goto label2;
        . . .
        statementN
    }
. . .
label2: statement
```

Although a while statement is shown, such control passing can occur within any conditional or loop statement. Some languages do not allow practices of this kind. In particular,

transferring control into the flow statement is generally not allowed. Since the use of goto's is discouraged, this attribute is not as important as the next two.

The second point typifies C flexibility. Many languages require the test condition to be a comparison test. As has been shown, the object expression and statement in C's control flow statements may be of any valid type. Restrictions also occur regarding the type of object statement allowed. For example in many versions of FORTRAN, the WHILE-DO statement cannot be the object statement in a DO loop; nesting of this sort is disallowed. Once again C demands no special considerations other than the inclusion of braces to form a compound block.

Lastly loop variables are treated no differently from others of the same data type and class. In FORTRAN, PL/I, BASIC, as well as other languages, some loop variables (notably the test and the increment values) cannot be altered. Furthermore these variables and their values are often denied to other statements. In general no such restrictions occur with C's loop variables, although when such privacy is required, it can be obtained by the use of locally declared automatic variables (chapter 7).

The break Statement [Advanced]

Sometimes a programmer wants a loop or a switch statement to be exited prematurely when conditions other than those outlined in the test condition occur. The break statement causes an early exit from the immediately enclosing while, do-while, for, or switch in which it occurs. A simple break statement is composed of the reserved word **break** followed by a semicolon. However, break statements are almost always (one of) the object statements of a conditional statement or one of the operands in a conditional expression, ?:, because unconditional execution of a break would reduce the

loop to a plain sequence of statements. **break** statements are illegal outside of loop or **switch** statements.

Generally the use of **break**'s to terminate control flow statements can be divided into four related categories:

- for loop termination due to error detection or other extraordinary circumstances
- as a supplement to the format test condition to a loop
- to terminate an otherwise infinite loop
- to terminate a loop or switch statement partway through, thus bypassing the execution of subsequent object substatements

The first two uses are closely related; both encourage readability and allow for loop testing whenever convenient. Consider the following **while** statement:

```
while (num > 0 && num != 5 && num != 10 &&
       num != 15 && num != 20 && num != 25
       && num < 27 && num != 666)
    { . . .
    }
```

Although the test condition is still manageable in this example, it still can be better coded by logically dividing the comparison operations into groups, then moving the more specific test comparisons into the compound object substatement. This can be accomplished through the use of the conditional **break** statement (i.e. the if conditional with a simple **break** object statement). This **while** statement can be recoded as follows:

```
while (num > 0 && num < 27)
    { if (num == 5 || num == 10 || num == 15
          || num == 20 || num == 25)
          break;
      if (num == 666)
          { printf(\n\t **ERROR**");
            break;
          }
    }
```

The more specific exceptions to the general loop condition are included within the body of the object block. The reverse condition must now be used in the **break**'s because a loop termination rather than a loop execution condition is being tested. Additional if substatements other than the **break** can also be executed, as exemplified by the printf() error message in the second if substatement. A similar if statement, without the **break**, would have had to have been coded after the original while construction to obtain the same error output. (Sometimes splintering of related code in this manner can cause problems in a program's cohesion and readability.)

The continue Statement

The **continue** statement is the compliment of the **break** statement. The continue statement can only be used within loops, as a substatement for the while, do-while, or for loops. When a continue statement is executed, control is passed immediately back to the test condition in the loop control line (in the for loop, expression3 is executed before control transfer). Therefore when a continue command is executed, all subsequent substatements are not executed for that loop repetition.

Unlike break, the continue keyword cannot be used in switch statements. The continue keyword is inevitably situated within a conditional substatement.

Consider the following short program:

```
main()

{   int a;
    for (a = 1; a < 11; ++a)
        { printf("%d    ",a); /* 5 */
          if ((a % 2) == 0)
              continue;
          printf("\b\b\b*    ");
        }

}
```

The for control statement is a classical one. The index is initialized to one and is incremented by one at the end of each loop repetition until **a** reaches 11. In each repetition, the value of the index and three blanks is output (line 5). Then if the result of a modulus division by two is zero (i.e. even number), a continue command is implemented and control passes back to the test expression.

Note that in these instances **a** is not reinitialized to a value of integer one. If the index **a** is odd, then the if condition of line 6 evaluates to false and the continue substatement is ignored. Then control passes to line 8, which outputs an asterisk after the most recently printed number. (The \ **b** escape sequence denotes a backspace.) After this statement, the brace on line 9 is encountered and control passes to the condition expression of the for. As a result of this construction, all odd numbers are output with a asterisk immediately ensuing:

```
1*   2   3*   4   5*   6   7*   8   9*   10
```

The conditional continue construction is equivalent to an if expression with an opposite condition. The previously presented program can therefore be recoded as follows:

```
main()

{   int a;
    for (a = 1; a < 11; ++a)
        { printf("%d   ",a);   /* 5 */
          if ((a % 2) != 0)
              printf("\b\b\b*   ");
        }
}
```

Nested Loop Constructions

Often it is advantageous to repeat a process which is itself iterative. These circumstances beg the use of *nested loops*, or the containment of one loop within another.

Consider the situation where one while loop is contained

within another **while** loop. The nested loop would be the simple substatement, or one of substatements comprising the object compound statement. The symbolic code for a nested **while** loop construction is shown in figure 5.14.

Assume condition1 initially evaluates to true, stipulating that the entire brace-delimited subblock is to be executed. As expected execution starts with statement1a and progresses (assuming that no **goto**'s exist within this entire block) until the nested **while** is encountered. Then if condition2 is true, statements 2a and 2N are executed.

Control then passes back to condition2 which is re-evaluted to determine its truth value, and the process starts over. Otherwise if condition2 evaluates to false, then the inside while **loop** is terminated and control passes to the statement immediately after the current loop. In this case, statement1K is executed after the nested **while** is finished.

Control then continues in a downward manner until the closing brace located on the last line of this section is encountered. Since this brace denotes the end of the encompassing object statement, control is passed to the first line and condition1 is reevaluated. If it is true the process starts over, but if it is false the entire (outer) **while** is terminated and control passes to "next statement".

This type of construction is referred to as a *second-order loop* or more commonly as a loop that is *nested one deep*. If one of the substatements within the nested **while**, such as statement2b, was itself some type of loop, then the construction would be nested two deep. It would be a third-order loop.

On the other hand, if one of the statements of the same hierarchical level as the nested **while** were to be a loop statement, then the construction would still only be second order. In that case, two separate, non-nested loops would both be nested within an all encompassing mother loop. A loop must be part of the object block of another loop to be nested within it.

```
while (condition1)
    (statement1a
    statement1b

      .  .  .

      while (condition2)
          (statement2a
          statement2b

          .  .  .
        } statement2N

    statement1K
      .  .  .
    statement1N
    }
next statement
```

Figure 5.14. Symbolic code section for while loop nested within a similar loop.

```
                /* Program 5.11 */
/* Accepts integer input, outputs factorial */
main()

{ char answ = ' ';            /* 5 */
  int in_num, count;
  long fact;
  do
    { printf("\nFactorial wanted? Y-yes, N-no\t");
      do                                              /* 10 */
        scanf("%c",&answ);
      while (answ == ' ' || answ == '\t' || answ == '\n');
      if (answ == 'y' || answ == 'Y')
        { printf("\nEnter integer < 17 \t");         /* 15 */
          scanf("%d",&in_num);
          for (fact=1, count=1; count != (in_num + 1); count++)
            { fact *= count;
              printf("\ncount=%d, fact=%ld",count,fact);}
          printf("\n%d! equals %ld",in_num,fact);      /* 20 */
        }
      printf("\n");
    }
  while (answ == 'Y' || answ == 'y');
}
```

program output on next page

```
Factorial wanted? Y-yes, N-no    y

Enter integer < 17    3

3! equals 6

Factorial wanted? Y-yes, N-no    Y

Enter integer < 17    12

12! equals 479001600

Factorial wanted? Y-yes, N-no    n
```

Figure 5.15.   A program exemplifying the practical application of the nested loop construction.

A practical example of the use of nested loops is presented in figure 5.15. Program 5.11 inputs as many integer values as the user signifies (one at a time) and produces the factorial for each. The factorial operation is symbolized by an exclamation mark. This mathematical operation is defined by the equation:

$$n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n$$

The majority of the body of program 5.11 is one do-while loop. This statement is in turn comprised of four sub-statements:

- An initial prompt printf() on line 9
- A second order do-while loop on lines 10 through 12
- An if conditional on lines 13 through 19
- A printf() "spacer" on line 20

Note the for loop of lines 16 and 17 is contained within the if substatement of lines 13 through 19. Because an if statement is conditional and not iterative, it is not correct to refer to the loop beginning on line 16 as being a nested loop (at this level). The for is, however, nested within the parent do-while statement. It is therefore also a second order loop.

The outside do-while loop is controlled by the condition on line 22. When the variable **answ** has a value of y or Y, the entire loop is repeated. This variable is initialized on line 5 to a character value of blank. Because a do-while condition is not tested until after loop execution, lines 9 through 20 are always executed at least once.

Line 9 prompts for a character answer. Because scanf() does not skip whitespace when a character conversion is called for, the program must include a device to assure that the input does not contain leading blanks or that these blanks are ignored. A message such a "Don't skip spaces before the answer — " could be output but this is inflexible, confusin and sloppy programming. Instead the do-while loop was us

to skip over whitespace.

Line 13 specifies that if the answer to the prompt was Y or y, then lines 14 through 18 are to be executed. Lines 14 and 15 are the integer input prompt and assignment statement respectively. Line 15 begins the for statement where all of the actual mathematical operations are performed. Object expression1 initializes both variables **fact** and **count** to one, expression2 is the loop execution condition, and expression3 states that variable **count** is to be incremented after every loop execution.

The object statement on line 17 is a simple statement. It multiplies **fact** by the current value of **count**. Because of the classical construction of this loop, the loop will be executed a number of times equal to the input value. For example, line 18 outputs the original input value and its factorial. An input value of 13 or less was specified because on the reference system used with this text, a value of 13! caused an overflow condition resulting in an incorrectly calculated factorial.

The condition of the parent do-while loop is finally checked on line 22. If the original answer was other than a Y or y, then the if block would not have been executed and the do-while would terminate at this point. If the original answer was affirmative, then the factorial calculation and output would proceed as described, and the parent loop would be repeated.

The goto Statement [Advanced]

goto is the only control flow statement that does not have an inherent conditional test. The goto statement follows the syntax:

goto *label*;

where the label is an identifier that must follow the rules of identifier name composition. Labels are not declared in the

way that variables are. Instead they are declared when utilized. Any statement may be preceded by a label identifier immediately followed by a semicolon:

label: statement

Accordingly when a goto statement is executed, control passes to the statement prefixed by the goto-specified label. Then flow progresses as prescribed by the labelled statement.

While goto statements are not loop statements per se, loop constructions can quite readily be created utilizing them. Consider program 5.12 in figure 5.16. It contains three labels: **loop, term1**, and **term2** and coincidentally three goto statements: two conditionals on lines 13 through 16 and a unconditional goto on line 17.

The unconditional goto forms an otherwise infinite loop by directing control to a previous statement. The conditional goto's act as conditional **break** statements would in normal loops — allowing control to pass from the loop when the stated conditions are met. They differ from **break**s in that they can direct control wherever their label appears in the function. They could, in fact, direct control upwards to form another loop that shares statements with the unconditional loop. Also goto loops do not demand compound blocking as do other control flow statements, although enclosing braces can be used if desired.

The program divides variable **b**, whose initial value is one, by successively larger integer values. The loop terminates when either the divisor exceeds 8 or when the quotient falls below exp(-10). If the latter occurs, control will pass from line 16 to line 18 and a underflow message will be printed.* In the former case, flow is passed to line 19, which is a null statement.

---

* Actually the way to program is constructed, the quotient can underflow on the last repetition without causing such a message since the condition b>8 is checked first.

```
/*    Program 5.12 - demonstrates the use of
  the unconditional and conditional goto
  statements to construct loops as well
  as "jump" control flow constructions
            *NOT RECOMMENDED*          */
main()

{  int b = 1;
   double a = 1;
 loop: a /= b;          /* 10 */
   ++b;
   printf("\na = %1e  \tb = %d",a,b);
   if (b > 8)
       goto term2;
   if (a < 1e-10)      /* 15 */
       goto term1;
   goto loop;
 term1: printf("\nUnderflow");
 term2:    ;
   printf("\nFinal values  a = %1e,",a);
   printf("    b = %d",b);
}                       /* 22 */
```

```
a = 1.000000E+000       b = 2
a = 5.000000E-001       b = 3
a = 1.666667E-001       b = 4
a = 4.166667E-002       b = 5
a = 8.333333E-003       b = 6
a = 1.388889E-003       b = 7
a = 1.984127E-004       b = 8
a = 2.480159E-005       b = 9
Final values  a = 2.480159E-005,    b = 9
```

Figure 5.16. Program illustrating the employment of both unconditional and conditional goto's.

After control is passed to a labelled statement, control flow precedes normally thereafter. For example, if an underflow situation was detected by the condition on line 15, then control would be passed via the object **goto** statement on line 16 to line 18. Because line 18 is a normal sequential statement, after it is executed, flow passes to the next line.

The next line, line 19, is the null statement. It is "executed" next. The label **term2** in no way affects execution of a statement. After the statement on line 19 has been executed, line 20 is executed. Observe that the statement in line 20 could have been written on line 19 after the label, since both conditional **goto**'s would still direct control flow to it.

Any type of statement may be labelled. It is illegal to have more than one label per statement. Such duplicity is unnecessary in any case since more than one **goto** statement may direct flow to any single label. Remember because labels must follow the rules of identifiers, they cannot duplicate any keywords. Labels must be unique identifiers with respect to the function in which they occur.

The use of **goto** statements is discouraged. Although program 5.12 is relatively orderly, the use of **goto**'s can easily undermine the logic and readability of a program. Worse still, when found within decision or loop statements **goto**'s change the standard control flow properties of these constructions, just as they ruin the general, overall sequential execution pattern of statements within a program. Some programmers maintain that **goto** statements have no place in structured, high level programming.

The Reference Program: The Final Version

Nearly all of the elements required to code the complete version of this chapter's reference program have been discussed: decision and loop constructions, compound relational tests, **scanf()** input capabilities, nesting of control flow state-

ments, data validation, and general program design criteria. If he or she has not already done so, the reader is advised to read the sections on the **goto** and **break** statements, as both of these are used in the final version of the reference program — program 5.13 is figure 5.17. Also before a detailed discussion on this last version is launched, the reader must be introduced to the concept of returned values.

scanf() RETURNED VALUES

There are several instances in program 5.13 where an expression of the following form occurs:

if (scanf("*control string*" **&***var1***...) !=** *constant***)**

Because the scanf() function call appears as the first operand of the inequality operator, it must be resolvable to an [extended] fundamental data type value. Up to now this function has only been used to assign input values to the variables pointed to in its argument list.

This library routine also performs the associated duty of returning the number of fields successfully converted. One can think of this value as replacing the entire scanf() call in the condition. For example if the scanf() call on line 55:

if (scanf("%d ,%f"**&st_num,&score) != 3)**

successfully converts three fields — a decimal integer, a comma, and a floating point decimal value in that order, then it will return a value of integer 3, and the condition will evaluate to false. Conditions of this form are utilized throughout the program to determine whether all argument variables have been correctly assigned values by scanf().

```
                    /* Program 5.13 */
  /* Final version of chapter 5 's reference program */
main()

{  char grade, ans = 'Y';
   unsigned test, er_fl = 0, tot_pts, audit;
   long int st_num;
   float score, per_sc;
       /* start of test # and total pts loop */
test_num:    ;                          /* 10 */
   if (er_fl == 1)
       { scanf("%*s");
         er_fl = 0;
       }
   printf("\nEnter test number - ");
   if (scanf("%d",&test) != 1)
       er_fl = 1;
   printf("\nEnter total number of points-  ");
   if (scanf("%d",&tot_pts) != 1)
       er_fl = 1;                      /* 20 */
       /* error message switch statement */
   do
     {
       switch (er_fl)
         { case 0:  break;
           case 1:
             printf("\nUnnacceptable test number");
             printf(" or total points");
             goto test_num;
           case 2:                     /* 30 */
             printf("\nIncorrect student number");
             printf(" or test score");
             break;
           case 3:
             printf("\nInvalid audit response");
             break;
           case 4:
             printf("\nInvalid negative score input");
             break;
           case 5:                     /* 40 */
             printf("\nScore larger than total ");
             printf("points possible");
             break;
         }
         /* retry message and data stream clear */
       if (er_fl != 0)
           { if (er_fl != 5)
                 scanf("%*s");
```

```
      printf("\nRe-input last group of data");
      er_fl = 0;                 /* 50 */
    }
   /* student values input and checking */
printf("\nEnter student number, (comma required)   ");
printf(" test score \n");
if (scanf("%ld , %f",&st_num,&score) != 3)
    { er_fl = 2;
      continue;
    }
printf("\nEnter 666 if audit, zero otherwise-  ");
if (scanf("%d",&audit) != 1)        /* 60 */
    { er_fl = 3;
      continue;
    }
audit = (audit == 666) ? 1 : 0 ;
if (score < 0)
    { er_fl = 4;
      continue;
    }
if (score > tot_pts)
    { er_fl = 5;              /* 70 */
      continue;
    }
      /* grade calculation */
per_sc = (score/tot_pts) * 100;
if ((per_sc >= 0) && (per_sc <= 60))
   grade = 'E';
else if ((per_sc > 60) && (per_sc <= 70))
   grade = 'D';
else if ((per_sc > 70) && (per_sc <= 80))
   grade = 'C';                         /* 80 */
else if ((per_sc > 80) && (per_sc <= 90))
   grade = 'B';
else if ((per_sc > 90) && (per_sc <= 100))
   grade = 'A';
  /* student grade output */
printf("\n\t\t\t\t\tTest #%d",test);
printf("\n  Student number- %9ld    Score- %5.2f "
                              ,st_num,score);
printf("   Grade-%c",grade);
if (audit)
    printf("\n\t\t\t\t\t * AUDIT *");
printf("\n");                       /* 91 */
    /* additional data prompt */
        /* and input */
printf("\nMore input values  ( Y or N )-  ");
do
```

program continued on next page

```
         scanf("%c",&ans);
      while (ans != 'Y' && ans != 'y' &&
             ans != 'N' && ans != 'n');
   }
   /* end of parent do-while loop */    /* 100 */
   while (ans == 'Y' || ans == 'y');
   printf("Total points possible -  %d\n",tot_pts);
}
```

Figure 5.17. Final version of a program designed to deal with this chapter's reference program.

```
Enter test number - 1

Enter total number of points- 250

Enter student number, (comma required)   test score
526219488 238.667

Incorrect student number or test score
Re-input last group of data
Enter student number, (comma required)   test score
526219488 ,  348.667

Enter 666 if audit, zero otherwise-   0

Score larger than total points possible
Re-input last group of data
Enter student number, (comma required)   test score
526219488   , 238.667

Enter 666 if audit, zero otherwise-   5

                                   Test #1
   Student number- 526219488    Score- 238.67    Grade-A

More input values  ( Y or N )-  y

Enter student number, (comma required)   test score
123456789 , 122.45
```

```
Enter 666 if audit, zero otherwise-  666

                                        Test #1
      Student number- 123456789    Score- 122.45    Grade-E
                                           * AUDIT *

      More input values  ( Y or N )-  n
      Total points possible -   250
        ;
       ·)
```

Figure 5.17a.   Trial run of program 5.13.

On lines 12 and 48, scanf()'s also appear with the control string "%*s". The suppression control character, *, has been introduced, but the control character, s, calls for a field conversion to a string. A string is an arbitrary number of characters. These two scanf()'s effectively skip one group of consecutive non-whitespace characters.* The %s will be more fully documented in chapter 9. These scanf()'s are executed only after an input conversion error has occurred. They are used to clear the input stream of one incorrect data string. This process is necessary because when scanf() encounters a conversion mismatch, it "pushes" the mismatched field back into the input stream. This same data field will be waiting to be read by the next scanf() statement, so it must be purged.

If the reader has some doubts as to the value of this process, compile and run the following program using a non-numeric input value(s).

```
main()

{   int in;
    while (scanf("%d",&in) != 1)
        printf("\tMISMATCH");
}
```

---

* Like the numeric conversion characters, the string conversion character skips leading whitespace. It terminates conversion upon encountering the first whitespace character after the string.

Try it again after inserting a string suppression scanf() before line 5.

REVIEW OF PROGRAM 5.13

Since we have already discussed many of program 5.13's code sections, we will not attempt to make the review that follows comprehensive.

As always, the first program lines declare variables and initalize the error flag to zero. Line 10 is the object label of a conditional goto that, as shall be shown, is tripped after the error flag, er_fl, has been assigned a value of one. For now we will ignore the next three lines. Lines 15 through 20 are responsible for inputting the test number and total points, as well as for the error checking associated with these scanf() routines.

The parent do-while loop begins on line 22. Its first substatement is a switch statement that handles error messages. The error flag can assume several "on" values, each associated with an exclusive message.

Breaks are used to stop fall-through case execution except for case 1:. This case is tripped by an error flag set to a value of one by either line 17 or 20. After an appropriate error message has been output, control passes out of the loop to the labelled null statement (line 10). Control then passes to the previously ignored conditional on lines 12 through 15. This conditional now must evaluate to true. Consequently, the mismatched string is suppressed, and the error flag is reset to zero. The loop from lines 10 through 29 is repeated until valid values are input.

A conditional is located immediately after the switch. This conditional outputs a "retry" message and resets the error flag to zero if it was originally on. This conditional if also contains a nested if that suppresses an input string when er_fl has a value between one and four inclusive. Error five is not associated with an input read error. It therefore, does not require this stream field suppression.

The beginning of the next section of code, from lines 51 to 63, handles the other three input values in a similar way as the first test values except continue's are used to direct control back to the error handling switch. Line 64 contains a conditional expression which changes the value of **audit** into a simple true or false value. Then two if statements check for scores that are less than zero or greater than the maximum possible, respectively.

The next section assigns a letter grade to the input score. The only new addition is line 74 which calculates a percentage score. Percentage scores are then used, rather than straight scores, to determine grade assignments.

Six lines responsible for normal student report output then follow. Since error warnings are not required in the normal output, the far right side has been reserved for an audit message.

The last statements in the parent do-while loop prompt and input one character value. Not only is all whitespace ignored, but all values other than Y, y, N, or n are rejected by the nested do-while occupying lines 95 through 98. The parent loop ends on line 101. The loop is terminated by an "en" character reply. A final reminder of total points is output once, at the end of the program. Several input passes are shown in figure 5.17a.

COMMENTS ON PROGRAM 5.13

This program could be improved in several areas depending upon the programmer's preferences. The goto/label arrangement could be regarded as inappropriate. It could be replaced by a another degree of nesting or by a repetition of lines 11 through 20, inserted into **case 1:**. Secondly the entire switch statement could be more logically placed after the section headed "student values input and checking" if the break's within the cases were substituted with continue's.

You, the reader have probably noticed that this is by far the longest program presented thus far. Despite the definition of subsections through the use of comment lines, programs of considerable length tend to be confusing, especially when many different problem aspects are handled. The next chapter will present the concept of multiple user-defined functions and argument passing. With these tools, it is possible to break down a program like program 5.13 into smaller modular units. This coding technique has two advantages. The program is divided into logically simpler units that improve readability. Since these same units are reusable, they often improve source coding efficiency.

6

Functions

Introduction

Up to this point programs have only consisted of the one user-defined (i.e. main()) function. Programs of considerable length, such as the final version of the reference program of chapter 5, are usually comprised of several user-defined functions, one of them always being main(). This chapter presents the concepts underlying multi-function programs.

While it is possible to code any program utilizing one function, there are a number of serious drawbacks to such an approach. First, there is a limit to the source file length which can be compiled under any computer system. This factor is especially critical with microcomputers. If, during compilation, the object or any intermediate code exceeds the memory space available to it, the compiler routine will normally automatically abort. Under these circumstances, the programmer must either seek out a larger computer system or divide the program into smaller modules, which are called *functions* in C.

This subsectioning approach is also clearly superior in terms of program development. Most programmers rely on a technique of programming called "*top-down development*". In this programming style, the high level logic of the overall problem is solved first while the details of each program section are addressed later. This style is implicitly assumed during flowchart interpretation of an algorithm. Flow diagram process rectangles contain broad descriptions of the actual set of statements needed.

A top-down approach is more directly translatable to actual source code when a program is divided into the driving main() function and specific, called functions. Because a function call is symbolic of a much more complex process, main() can be made more concise and graphically closer to the higher logic "skeleton" of the program's flow diagram by the implementation of auxiliary or supplementary (i.e. non-main()) functions.

A subsectioned program is also easier to debug. Many errors can be traced to the operation of one or several functions. These faulty functions can then be isolated and run by themselves through the use of a *test driver*. A test driver is a simple version of main() designed to test the operation of specific functions. Because of their modular, independent design, functions are much easier to isolate than code sections of the alternative one function program.

Because the use of a pre-defined function hides unnecessary details of a process, the programmer is not required to understand how a function or a routine (i.e. group of related functions) operates, but only what the call accomplishes. Many routinely-used functions operate using this "black box" principle. For instance, the scanf() and printf() functions have been used in this manner throughout the text. These library routines are actually quite complex and system dependent. Fortunately most programmers need not know exactly how these routines operate; it is sufficient that they operate as expected. Functions then allow a C programmer to build upon the knowledge and skills of other programmers without duplicating their efforts.

Not only can functions be easily isolated, but they can be easily recombined for use in other programs. In other words, specific functions are transportable. Auxiliary functions can either be explicitly linked to the object program (e.g. as library routines normally are) or they can be included in the main() source file. Both of these methods allow reuse of functions and routines among different programs. (More will be presented on compilation and related topics in chapters 8 and 11.) The ability to isolate functions also increases the convenience of file storage.

The Form of C Functions

All functions have the form:

```
type-specifier identifier (formal argument list)
formal argument declaration;
...
{ body declarations ;
  :
  other statements
  ...
}
```

The main() functions presented thus far have not included an (explicit) type-specifier. There also has been no declaration of formal arguments because the formal argument list has been null. Therefore the following function:

```
main()

{   int a, b = 1, c = b;
    flt x, y;
    statement1
    statement2
    . . .
}
```

has **main** as its identifier, a null formal argument list, two lines of body declarations, and an arbitrary number of other statements.

The following sections detail the requirements for each component of the function header. The beginner is advised to skip over to the section "The Development of the Concept of Multiple Functions" and reserve the ensuing sections for later.

TYPE-SPECIFIER

The type-specifier identifies the data type of the value returned to the calling function. The allowed data-specifiers, corresponding to the [extended] fundamental data types, are: short intopt, unsigned intopt, long intopt, [unsigned] char, float, long float, and double. However, since short int and char types are automatically promoted to int and likewise float's are promoted to double (with long float usually being equivalent to double), the type-specifiers int and double are normally used. In fact some inexpensive compilers automatically perform type-specifier promotion. The default type-specifier is int. If a function returns a derived data type (appendix F), then the type-specifier must be that derived data type.

The type-specifier prefixing the function must match the type-specifier used to declare that same function in the body of the calling function or a type mismatch will occur, resulting in a compilation time error.

FUNCTION IDENTIFIER

The function name must follow the same rules of formation as other identifiers as presented in chapter 2. Additional care must be taken to avoid duplicating library routine names or operating system commands. Also name length may be subject to further restrictions imposed by the operating system and/or assembler (see chapter 8).

FORMAL ARGUMENT LIST AND DECLARATIONS

The argument list must be surrounded by parentheses. By convention the opening parenthesis is positioned immediately after the function name. The formal argument list contains the variables that receive copies of the actual arguments in the calling statement. (Formal argument variables are also referred to as the parameters of their respective function.) Both formal and actual arguments must be separated from each other in a list by commas. Variable names may match across functions but no inherent relationship exists for automatic variables. The number of formal and actual arguments need not coincide, although unpaired formal arguments are guaranteed to contain garbage. No semicolon follows the closing parenthesis of the formal argument.

All formal argument variables must be declared after the first header line and before the opening brace of the body. Argument data types need not correspond with the type-specifier of the function, but any actual to formal argument data type mismatch is likely to result in garbage copied values.

CALLED FUNCTION DECLARATION

Any function that is called from within the body of the current function should be declared within the body of the current function. Declaration must take place before the non-declaration statements of the block or parent block in which the corresponding function call occurs. Function declarations, by convention, occur before local variable declaration, and have the form:

type-specifier identier1(), *identifier2*(), ...;

Functions that are not explicitly declared and are not defined before the calling function have a default int type. Mismatch between the type-specifier used in the declaration and that used in the header or a function will result in a compilation time error message such as "redefinition of function". Most compilers will handle calls to previously defined functions correctly without any function declaration in the calling function. This convenience is, however, highly correlated with file reorganization errors.

The Development of the Concept of Multiple Functions

In this section, the concepts needed to produce programs with more than one user defined function will be developed. This section will begin with the most elementary sort of function interaction and gradually progress to the more general and complex. All the functions presented in this section use only integer arguments and return only integer values. Functions that handle non- int quantities are only slightly more complicated, and will be dealt with in the next major section.

The programs presented in the next few subsections will consist primarily of two functions: the mandatory main() and a second function named g_c_d(). This latter function in all its

incarnations performs the job of calculating the greatest common denominator (G.C.D.). The GCD of two (or more) integral numbers is the largest integer value that will divide into the two (or more) original operand values without leaving a remainder. The GCD of 30 and 12 is 6, while the GCD of 3 and 9 is 3.

NO ARGUMENTS AND NO RETURNED VALUES

The first version of the program is presented in figure 6.1. Ignore main() entirely for the time being. The function g_c_d() starts on line 11. Notice it has no explicit type-specifier, so by default, it is assumed to be an int function. Since it contains no formal arguments, there are no argument declarations before the function body. g_c_d()'s body looks much like a corresponding main() would. All variables are initialized on lines 13 and 14. A prompt and corresponding input follow on lines 15 and 16. Because the function will output these original values again, before manipulation it is necessary to copy the original values of the two input integers into the variables **gcd, big,** and **sml.** The **if-else** statement in lines 17 through 24 assigns the smaller of these two input values to both **gcd** and **sml,** while the greater value is assigned to **big.** Since the GCD cannot logically exceed the smaller of the two (or smallest of several) operand values, the smaller input value was assigned to **gcd,** the variable that will eventually hold the calculated GCD quantity.

The **for** statement of lines 25 through 30 actually performs the mathematical GCD operation. The condition expression corroborates that the input values are positive; if the lesser of the two is positive, both must be. Later, a version of g_c_d() will be introduced that properly handles negative inputs. The modulus operator is used on lines 26 and 27 to assign the remainder variables (**rmdr1** and **rmdr2**) the remainders of the division of the input values and **gcd.** The conditional **break**

that follows is tripped when both remainders are zero — the definition of a GCD. If it is not tripped, **gcd** is decremented and the loop reiterates. For two positive integer inputs, the smallest value attainable by **gcd** is one. All integers are evenly divisible by one.

Now take a look at the main() function in program 6.1. It normally begins with the declaration of all variables used in its body followed by an arbitrary number of statements which have been symbolically represented by the comment on line 6. The line of central interest is line 7, the function call to g_c_d(). Note that it has a null argument list; no information is transferred to the function named in the function call, g_c_d(). When line 7 is executed during a program run, control passes to the function g_c_d(). This function is then executed line by line as described. When the closing brace of g_c_d()'s body is reached (i.e. the called function has been executed), control passes back to the calling function, main(). main() then continues execution, starting at line 8 and executing "more statements" in the normal manner.

It is important to note that no communication of data has taken place between the two user-defined functions of this program. It is almost as if the program consisted of two separate subprograms that were almost entirely independent of one another. main() neither sends nor receives any information from g_c_d(). Only in the order of execution are they related.

Notice that both functions declare the integer variable **gcd**. It seems logical to assume from the discussion presented up to this point that once the variable **gcd** has been assigned a value by the calculation performed in the function g_c_d(), then the variable **gcd** in main() will have received this calculated value. This indicates that communication of information has taken place across functions. This assumption is incorrect. *The variables* **gcd** *in* **main()** *and* **gcd** *in* g_c_d() *are actually two unique and separate variables* as much so as if they had

different identifiers. The fact that these variables are unique arises from the concept of the automatic storage class.

```
              /* Program 6.1 */
/* Use of a isolated called function to find GCD */
main()

{  int oprd1, oprd2, gcd ;        /* 5 */
   /* other statements */
   g_c_d();
   /* more statements */
}
              /* 10 */
g_c_d()

{  int in_int1, in_int2, gcd;
   int rmdr1, rmdr2, sml, big;      /* 14 */
   printf("\n\tInput two + integers for GCD\n\t");
   scanf("%d%d",&in_int1, &in_int2);
   if (in_int1 > in_int2)
       { gcd = sml = in_int2;
         big = in_int1;
       }                            /* 20 */
   else
       { gcd = sml = in_int1;
         big = in_int2;
       }
   for ( ; sml > 0; --gcd)
       { rmdr1 = big % gcd;                 /* 26 */
         rmdr2 = sml % gcd;
         if ((rmdr1 == 0) && (rmdr2 == 0))
             break;
       }                                    /* 30 */
   printf("\nThe GCD of %d and ",in_int1);
   printf("%d is %d",in_int2,gcd);
}
```

```
       Input two + integers for GCD
          45  18

The GCD of 45 and 18 is 9
```

Figure 6.1. First version of a program for calculating the greatest common denominator (GCD) of two input integers.

AUTOMATIC VARIABLES AND THE PRIVACY
OF FUNCTIONS

It was stated in chapter 3 that all variables have a data type and storage class associated with them. Chapter 3 introduced the fundamental storage class and the extension "adjectives". Actually all of the variables presented so far also belong to a storage class termed *automatic*. Any variable declared within the body of a function or in the formal argument declaration before the body of the function will be of the automatic storage class unless declared otherwise. That is, automatic is the default storage class for variables located in the function header or body. These variables may be explicitly declared using the keyword **auto**.

Examine figure 6.2. In this program main() initializes **auto** variables **a** to decimal 5 and **b** to 7. Line 8 then executes a call to out(). In the called function, variables **a** and **b** are again declared as **auto int**'s on line 18. Their values are output after an identification header. Obviously the values of **a** and **b** output from out() have no connection with those assigned to the variables **a** and **b** in main().

Automatic variables are only visible within the function in which they are declared. Each automatic variable is said to be *private* to its corresponding function (or parent block). Program 6.2 actually contains four unique variables: main() has two **auto int**'s **a** and **b**, and out() likewise has two **auto int**'s **a** and **b**. The values output from line 20 are the garbage values that remained in the memory blocks when the declaration of line 18 occurred. After outputting these two values, out() increments **a** and **b** by 10 and 20 respectively. Control then passes back to main(). Lines 9 and 10 then increment the **a** and **b** of main() by 1. Then another call to out() occurs on line 11.

This call to out() duplicates the last call, however the outputted values are entirely different. What happened? When the previous call to out() in line 8 was executed, control only

```
/*      Program 6.2 - Two user defined
   function program illustrating the na-
   ture of auto variables           */
main()

{  auto int a = 5;
   int b = 7;   /* default auto */
   out();
   ++a;
   ++b;
   out();
   printf("\nfunction main()");
   printf("\na = %d, \tb = %d\n",a,b);
}

out()

{  int a, b;
   printf("\nfunction out()");
   printf("\na = %d, \tb = %d\n",a,b);
   a += 10;
   b += 20;
}
```

```
function out()
a = 94,    b = 160

function out()
a = 227,   b = 39

function main()
a = 6,     b = 8
```

Figure 6.2. Program illustrating the nature of automatic variables

passed back to main() after out() terminated execution — that is, when the closing brace of out()'s body was encountered. *Automatic variables cease to exist when the function (actually the parent block) in which they are declared terminates execution.* So upon this second call to out(), blocks of memory were again allocated to its variables. The new set of garbage data found in these blocks is shown to differ from the last.* The next chapter will introduce two storage classes that circumvent this attribute of function-dependent longevity.

After the completion of second call to out(), main() outputs the values of its variables, **a** and **b**. Note that even though control passes out of main() twice, the two variables of main() are not discarded because main() does not terminate until the closing brace on line 23. Until main()'s termination (which normally coincides with the end of the entire program), all of its automatic variables are still "alive"

ARGUMENTS BUT NO RETURNED VALUES

The next step in developing the notion of multi-functional use is the insertion of argument lists into program 6.1. With this addition, the calling function, main(), would be capable of sending values to the called function, g_c_d(). The argument list of the function call is composed of *actual arguments*, while the components of the called function's list comprise the *formal argument list* (see figure 6.3). The arguments are matched one on one starting with the first argument in each list. Unmatched trailing arguments from the actual argument list are discarded (i.e. not used), while trailing unmatched formal arguments are initialized to previous garbage values. (Un-

---

* This can be interpreted as meaning that either a group of physically differ-
ent memory blocks (i.e. ones with different memory locations) have been
allocated to these same identifiers or the same memory block has been used
in between out() calls. When this simple program is executed on a single-
user system, it is fairly common to receive the expected output: a=104,
b=180.

matched arguments occur when one list is longer than the other.) Typically, however, both argument lists are of equal size. Each actual argument can be any expression resolvable to [extended] fundamental data type or pointer value. The formal arguments must be variable identifiers.

actual argument list

fun(a_arg1, a_arg2 , . . . , a_argk);

matching

? see below

fun(f_arg1, f_arg2 , . . . , f_argN)

formal argument list
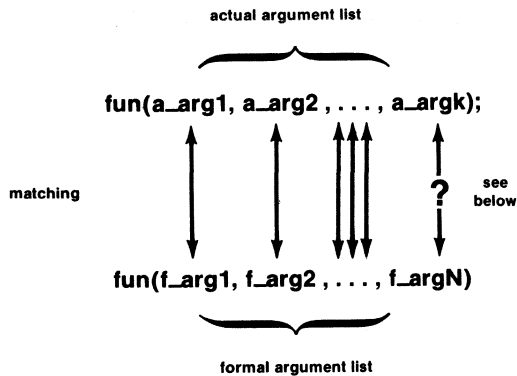
- if K equals N then a_arg1 through a_argk are matched (i.e. ←→) to f_arg1 through f_argN respectively.
- if K is greater than N then a_arg(N+1) through a_argk are unmatched and discarded.
- if K is less than N then f_arg(k+1) through f_argN are unmatched and initialized to garbage values.

Figure 6.3. Summary of argument matching between actual argument list of the function call (statement) and the formal argument list of the called function.

The second version of this section's reference problem on GCD's is presented in figure 6.4. Most of the program is identical to program 6.1 and will not be discussed again. The input prompt and **scanf()** assignment statement have been moved from g_c_d() to, lines 7 and 8 of main(). Because variables **in_int1** and **in_int2** are not "alive" in main(), the

input values must be assigned to variables that are global to
main(), namely **oprd1** and **oprd2**. The call on line 9 is now
more complex because main() must now pass the information
it contains to g_c_d(). The function call has two actual argu-
ments, and the function header on line 13 has two formal
arguments that are immediately declared afterwards on line 14.
Consequently **oprd1** is matched onto **in_int1**, and **oprd2** is
matched onto **in_int2**.

```
                /* Program 6.3 */
/* Use of passed data to 2nd function to find GCD */
main()

{   int oprd1, oprd2, gcd ;           /* 5 */
    /* other statements */
    printf("\n\tInput two + integers for GCD\n\t");
    scanf("%d%d",&oprd1, &oprd2);
    g_c_d(oprd1,oprd2);
    /* more statements */           /* 10 */
}


g_c_d(in_int1,in_int2)
int in_int1, in_int2;
                                    /* 15 */
    { int rmdr1, rmdr2, sml, big, gcd;
    if (in_int1 > in_int2)
        { gcd = sml = in_int2;
          big = in_int1;
        }                           /* 20 */
    else
        { gcd = sml = in_int1;
          big = in_int2;
        }
    for ( ; sml > 0; --gcd)
        { rmdr1 = big % gcd;                /* 26 */
          rmdr2 = sml % gcd;
          if ((rmdr1 == 0) && (rmdr2 == 0))
              break;
        }                                   /* 30 */
    printf("\nThe GCD of %d and ",in_int1);
    printf("%d is %d",in_int2,gcd);
}
```

program output on following page

```
        Input two + integers for GCD
           45  18

The GCD of 45 and 18 is 9
```

Figure 6.4.   Second version of the program for calculating the
GCD of two numbers. The calling function,
main(), passes both operand values to the called
function g_c_d(), which returns no value.

By matching, it is meant that a temporary, private copy of
the actual argument is assigned to the corresponding formal
argument. Therefore it may be helpful to view lines 9, 13, and
14 as accomplishing the assignments:

in_int1 = oprd1;
in_int2 = oprd2;

across function boundaries. (But remember since g_c_d()'s
variables are automatic, they will lose their values after the call
is completed.) Because this calling process passes only tempor-
ary, private values, it is often referred to as a *call by value*.

Aside from the absence of the input procedure, g_c_d()
executes exactly as before. It again calculates the GCD of two
positive integers and outputs the appropriate label and data.
Remember any change in the formal arguments' values does
not, in itself, affect the values of the corresponding actual
arguments.

ARGUMENTS WITH RETURNED VALUES

Now that communication with the called function has
been established, naturally the next question is "How does one
get information back from the called function?" The answer
to this question lies in the use of C's **return** statement. Through

the proper use of this statement, it is possible to send back to the calling function up to one value, but never more. *While it is possible to pass the called function any number of values, the called function can only return one value at most per call.*

This process is quite different than an extension of code (as PERFORM statements in COBOL allow). Function argument passing can be thought of as taking place within a pipe connecting only two functions at any one time.* An arbitrary number of copies of actual values are sent over this pipe to the called function. When the called function has finished executing, it is able to send up to one copy of a value back to the calling function through the same pipe. The pipe is now free to connect any two other functions together when and if another function call is encountered. Aside from this pipe, functions normally have no way of communicating with each other. (Actually there are two other methods of communication available to functions: external storage class variables, chapter 7, and pointers, chapter 9.)

The final version of this section's reference problem, presented in figure 6.5, illustrates the use of values passed both to and from called functions. There are two major changes in it over the last version, program 6.3.

One change is the movement of the output message from g_c_d() to main(). Since the printf() is now found in main() (lines 11 and 12), any variables it uses as arguments must be alive in main(). For the integer inputs, this simply means substituting the in_int's with their oprd counterparts (these values are constant). The actual value of the GCD, which was calculated in g_c_d(), must be passed to main() before it can be output from there. Line 37 accomplishes this task with the following statement:

return (gcd);

---

* The use of the word "pipe" in this analogy should not be confused with its proper use in operating system terminology, although the two are conceptually similar.

```
                    /* Program 6.4 */
    /* Use of a passed data and returned values */
    /* to find GCD utilizing called functions */
    main()
                                    /* 5 */
    {  int oprd1, oprd2, gcd_m ;
       /* other statements */
       printf("\n\tInput two + integers for GCD\n\t");
       scanf("%d%d",&oprd1, &oprd2);
       gcd_m = g_c_d(oprd1,oprd2);      /* 10 */
       printf("\nThe GCD of %d and ",oprd1);
       printf("%d is %d",oprd2,gcd_m);
       /* more statements */
    }
                                    /* 15 */
    /* finds greatest common denominator of 2 int's */
    g_c_d(in_int1,in_int2)
    int in_int1, in_int2;

    {  int rmdr1, rmdr2, sml, big, gcd;   /* 20 */
       in_int1 = abs_int(in_int1);
       in_int2 = abs_int(in_int2);
       if (in_int1 > in_int2)
           { gcd = sml = in_int2;
             big = in_int1;      /* 25 */
           }
       else
           { gcd = sml = in_int1;
             big = in_int2;
           }                     /* 30 */
       for ( ; sml > 0; --gcd)
           { rmdr1 = big % gcd;
             rmdr2 = sml % gcd;
             if ((rmdr1 == 0) && (rmdr2 == 0))
                 break;          /* 35 */
           }
       return (gcd);
    }
```

```
/* returns absolute value of single integer */
abs_int(abs)                    /* 41 */
int abs;

{   abs = (abs < 0 ? -abs : abs);
    return (abs);               /* 45 */
}
```

```
        Input two + integers for GCD
        45  -18

The GCD of 45 and -18 is 9
```

Figure 6.5. Third and final version of a program to calculate the GCD of two integers. Both passed argument values and returned values are utilized.

The value of **gcd** at this point is passed back to main(), where it substitutes for the entire function call that originally activated g_c_d(). In this way, variable **gcd_m** is assigned, on line 10, the value of **gcd** that passed back from line 37.

The following events occur, in order, when line 10 is executed:

1. The function call **g_c_d(oprd1, oprd2)** is encountered. Control then passes to the function **g_c_d** and copies of the values of the actual arguments **oprd1** and **oprd2** are assigned to their ordinally-matched counterparts in the formal argument list (line 17), **in_int1** and **in_int2**, respectively.

2. The called function g_c_d() is executed line by line in a normal fashion until a return keyword or the closing function body brace is encountered. In this example a return is encountered on line 37.

3. Control along with possibly one value, designated by the object expression of the return statement, passes back to main(). Any returned value is substituted in place of the original function call. This value is the *resolved value of the function call*. Here, the value of variable **gcd** is passed to and substituted for the call in line 10, leaving the equation:

$$\textbf{gcd\_m} = value\ of\ gcd;$$

4. The calling statement is executed normally, utilizing the returned value. Line 10 indirectly assigns **gcd_m** the value of **gcd**.

Note that the variable identifier **gcd_m** in main() was changed from its old name of **gcd**. This was done to avoid confusing this identifier with the duplicate identifier in g_c_d(). This was technically unnecessary because both are **auto** variables. It may also have been excessively cautious from a conceptual standpoint as both variables represent the same quantity. Many beginners have a tendency to mechanically duplicate actual argument names with those used in the formal argument. Experience has shown that this habit is associated with errors, as it can rapidly generate quite confusing code (not to the compiler but to people).

The second and most obvious change is the addition of the function abs_int(), whose purpose is to take one integer passed value and to return its absolute value. This function is called in lines 21 and 22. abs_int() is separately sent copies of the value of **in_int1** and **in_int2** which in turn hold copies of the originally assigned input variables **oprd1** and **oprd2** from main(). This short function returns this same value if the passed value is zero or greater, but returns the negative of the passed value if the original value is less than zero. (The unary minus operation on a negative value yields a positive value.)

These returned values are "substituted" for the entire function calls in lines 21 and 22. Here these values are reassigned back to the original actual argument variable to effect the desired change. The reassignment is necessary as the func-

tion call itself does nothing to alter the calling function variables' values. If **in_int1** equalled -12, then the operation of line 21 could be summarized by the following four steps:

Step 1:   **in_int1 = abs_int(in_int1);**   original statement
Step 2:   **abs = in_int1**   argument value copying
Step 3:   **abs = -abs;**   resolved conditional, line 44
Step 4:   **in_int1 = 12;**   returned value substitution

Through this statement the value of **in_int1** has been changed from -12 to 12.

It is critical that the reader understands that the function, in itself, did not effect this change. The function call to **abs_int( )** only represents a value. It is the assignment operator in line 21 that actually directs this change. Of course the resolved value of the expression on the right side of the assignment operator on line 21 could have been assigned to any variable other than **in_int1**. In this case, **in_int1** would still retain its initial value which was assumed to be -12.

THE RETURN STATEMENT

The **return** statement can take one of two forms in C;

return ;
or   **return (**opt *expression* **)**opt **;**

The first, the "plain" **return**, acts much as the closing body brace of the called function. When a plain **return** is encountered, control immediately passes back to "after" the point of the call in the calling function. Because plain **return**'s are equivalent to closing braces, they rarely appear as a simple statement, unless to emphasize the close of a function. Normally a plain **return** is (one of) the object statements of a conditional, as in:

if (error)
return;

The second form of the **return** statement has a similar function to the first, but it also sends the value of the resolved object expression back to the calling function. This value is "substituted" in place of the entire function call. Returned values are discarded after the calling expression is evaluated unless assignment to another variable is called for.

A function may contain more than one **return** statement. However since control passes back to the calling function when a return is executed, multiple occurrences of the **return** keyword are usually associated with conditionals:

```
if    (sum < 0)
        return ('D');
else
        return ('C');
```

INDEPENDENCE OF AUXILIARY FUNCTIONS

Programs 6.1 thru 6.4 obviously illustrate the concepts behind and implementation of argument passing and returned values. Just as importantly however, is the evolution of the g_c_d() function itself.

To assure a high degree of portability between programs, a function should generally be coded so as to be removed from any I/O operations, and independent (as much as possible) from the details of other functions or program environment. All program specific processes should be handled by main() and perhaps a few other auxiliary functions.

g_c_d() originated in program 6.1 as almost a separate subprogram. It had its own I/O calls and did not communicate any information to main(). The limitations of such a function are painfully obvious. This version of g_c_d() is generally worthless to the rest of the program as no communication exists between it and other functions.

Program 6.3's version of g_c_d() is superior to its predecessor because it does not specify the data source. Still it

contains a stated output operation (i.e. the printf()'s of lines 31 and 32). Obviously for a program that does not require this output, this function would be undesirable.

The final version of g_c_d() in program 6.4 is then the most superior. No assumption as to argument data organization or the destination of the returned value is made. The calling function could generate argument values from a scanf() input routine as was done in program 6.4, or the values could be the result of some internal data manipulation (such as mathematical operations). Likewise the returned value can be used or unused as determined by the calling function.

Self-contained and independent function usage also encourages a top-down structured approach as each auxiliary function or group of functions can correspond to a process in an algorithm.

As an aside, there are two additional objections to the use of printf() and scanf() function calls within reusable auxiliary functions.

- The C standard library can have other closely related I/O routines that may be preferred over the aforementioned routines.
- It is possible to link object code modules from C source code to object code modules from other languages. For such an attempt to succeed, calls to language specific external routines should be avoided (see chapter 8).

Functions Handling Non-integers

In the preceding multi-function examples, the type-specifier has been absent. The type-specifier precedes the function identifier and is used to specify the data type of the returned value. This absence was allowable because the default type-specifier for a called function is int. If desired, a function can be explicitly declared as type int. For example, the com-

plete called function header of program 6.1 would be written as:

Int g_c_d()

C was designed as a systems language, and integers are the data type primarily used. The explicit or implicit int declaration is also adequate for char and short int as these data types are automatically promoted to int.

Nevertheless, for applications usage, floating point or long integer processing is often indispensable. Refer to figure 6.6. The function abs_int() has been modified and renamed so as to handle floating point data passed to and from the attendant simple main() test driver. (It would be pointless to modify g_c_d() in a similar way since the GCD operation is meaningless when applied to floating point quantities.)

Two things must be changed to enable a called function to return a non-integer value:

- The explicit type-specifier corresponding to the data type of the returned value must be inserted prior to the function name in the function header.
- The called function must be declared at the start of the body in the calling function.

These steps are performed respectively in lines 17 and 6 of program 6.5. The default declaration in the calling function is, of course, int.

NON-INTEGER PASSED ARGUMENTS

Program 6.5, in addition to returning a non-integer, also passes copies of non-integer argument values. No extra procedure is necessary to enable non-integer argument passing, other than checking that the actual and corresponding formal arguments agree in data type. In general if argument pairs do not agree in data type, garbage will be passed without generat-

```
                /* Program 6.5 */
/* Modification of abs() to handle float-
   ing point flavored returned values */
main()

{   double abs();
    double inqu, a_inqu;
    printf("\nInput a decimal quatity: ");
    scanf("%lf",&inqu);
    a_inqu = abs(inqu);
    printf("\nThe absolute value of");
    printf(" %lf is %lf\n",inqu,a_inqu);
}

/* returns absolute value of single
   double precision quantity */
double abs(x)
double x;

{   x = (x < 0) ? (-x) : x ;
    return (x);
}
```

```
Input a decimal quatity: -345.12345678

The absolute value of -345.123457 is 345.123457
```

```
Input a decimal quatity: -45

The absolute value of -45.000000 is 45.000000
```

Figure 6.6. Example of a program containing a called function that returns a non-integer value.

ing an error message. The only data type conversions that occur between functions are those that are automatic: **short int** and **char** values are passed as **int**'s while **float**'s are passed as **double**'s.

The input variable **inqu** is declared **double** in **main()**. Because a copy of this variable's value is passed to **abs()** from the call on line 10, one would expect the formal argument variable to be also declared **double**, which indeed it is on line 18.

The returned value must be appropriate for the position for which it is being substituted. On line 10, the value returned by **abs()** is the operand in an assignment operation. The value returned will therefore be valid regardless of its data type because data type conversion will occur during assignment as required. On the other hand, if this same returned value had been used in a bitwise operation, an error would occur because bitwise operators require integral operands. This situation would require the use of the cast operator to demote the returned value to an integer flavored quantity.

```
mask = first I abs(inqu);      /*illegal*/
mask = first I (int)abs(inqu);      /*legal*/
```

Actual arguments must often be similarly cast so that they correspond to expected data types. Casting was not necessary in program 6.5 because all input values are automatically read in and stored as double quantities through the %1f conversion code on line 9 (see run #2).

Finally, remember that any function may be passed non-integer values without the added explicit declaration(s) and type-specifier inclusion previously described. It is only when the returned value of a function is non-**int** that these additional steps must be taken.

UNUSED PASSED VALUES

Sometimes a situation occurs where passed or returned values are not of interest. This normally occurs when predefined functions are utilized. Because these functions are designed with portability in mind, they may return information not relevant to the programmer's particular application. For example, the use of the value returned by scanf() for input data validation is completely up to the discretion of the programmer.

Any undesired leading actual argument values can be matched to non-utilized or "phantom" formal arguments that never actually appear in the body of the called function. Unwanted trailing actual arguments can likewise be matched to phantom variables, but a more common practice is to leave them entirely unmatched in the formal argument. All unmatched trailing arguments in the actual argument list are discarded without warning.

Function Call Usage and Call Nesting

A function call is a primary expression. Therefore it can be used wherever an expression is called for. With the addition of a trailing semicolon, a function call qualifies as a statement. When a function call appears alone as a statement, it usually performs a service associated with the operating system. For example, I/O routines can be invoked independent of any parent expression. Hence the statement:

printf(" \nHello");

is quite useful. In contrast consider the statement:

pwr(7,3);

where pwr() returns the value of its first argument raised to its second argument (i.e. 7 cubed is 343). Upon function evalua-

tion, the last expression becomes equivalent to:

343;

The last two statements are both quite worthless as the returned value is discarded after statement execution. The function call accomplished nothing outside of wasting time.

For this reason function calls are used quite extensively in operational (assignment) statements as has been demonstrated in the last section. Function calls can be advantageously utilized in other circumstances as well. For example, figure 6.7 demonstrates how a function call can replace a lengthy condition expression. Such a substitution can result in improved program clarity and a savings in coding time if the same condition is used more than once.

```
/* lengthly decision expression in main() */
main()

{   int a,b,c,d,e,f,g,h ;
    /* other declarations */
    /* other statements */
    if ((a > b || c > b) && ((d % f) == 0)
       || (c == (e = 100 * g) && (g != 0)
       && (f != 0) && (h != 1) &&
       ((a + b + c + d) < 500)
         /* if object statement */
    /* other statements */
}
```

```
/*  lengthy decision expression re-
     moved to separate function    */
main()

{  int a,b,c,d,e,f,g,h ;
   /* other declarations */
   /* other statements */
   if ( decl(a,b,c,d,e,f,g,h) )
       /* if object statement */
   /* other statements */
}

/* returns 1 if condition is true
        0 if false    */
decl(l,m,n,o,p,q,r,s)
int l,m,n,o,p,q,r,s;

{  if ((l > m || n > m) && ((o % q) == 0)
      || (n == (p = 100 * r) && (r != 0)
      && (q != 0) && (s != 1) &&
      ((l + m + n + o) < 500)
      return (1);
   else
       return (0);
}
```

Figure 6.7. The use of a function call and corresponding
function to replace a condition expression.
a) original main() with lengthy if condition.
b) replacement of condition with function call.

Function calls can also be nested in much the same
manner as control flow statements can be. For example, sup-
pose one wanted to find the greatest common denominator of
three numbers; 2169, 434, and 144. A function could be deve-
loped similar to g_c_d() that handled three quantities at once.
But then what would one do when faced with the job of finding
the GCD of four or more numbers? Luckily this operation can
be broken down into two simple steps:

- Find the GCD of any two of these numbers
- Find the GCD of the third number and the resultant value of the first step.

In code these steps become:

int1 = g_c_d(2169,434);

ans = g_c_d(int1,144);

where **int1** is an *intermediate* or *explicit temporary* variable.

When the intermediate value is not utilized for other purposes, these two steps can be combined by the use of a technique termed *function call nesting*. Thus the preceding two lines can be written as:

ans = g_c_d(g_c_d(2169,434),144);

The inner (most) function call must be evaluated first as the returned value is again used as an actual argument. This construction represents two sequential function calls, not chained or a recursive construction. This coupling should not come as a surprise since an actual argument is required to be an expression resolvable to a [extended] fundamental data type or pointer value, and function calls are primary expressions. C contains no inherent limitation as to the degree in which function calls or control flow statements may be nested. However each compiler imposes such a limit on practical grounds. Nesting orders of up to ten are usually achievable on even small systems.

Although function calls can readily be nested, functions cannot. It is illegal to have one function's header and body located within another function's body.

goto's IN FUNCTIONS [Advanced]

The goto statement and affiliated use of labels was discussed in chapter 5. The reader may wonder whether it is possible to develop an extension of normal code from one function to another. The answer is no. *A* goto *can only direct*

control to the corresponding labeled statement within the current function. (i.e. the function in which the goto is located).

Recursion [Advanced]

Function calls obviously act as control flow devices. Upon call execution control passes from the calling function to the called function. The called function is executed until a return or the closing brace is encountered, at which point control passes back to the point "after" the function call. Conditional return's can be used inside the called function to further vary control.

It has also been demonstrated that when a called function in turn calls another function a process of chaining occurs. A special instance of this process is termed *recursion*. Recursion is the process where by a construct operates on itself. In other words, a function is recursive if it directly or indirectly calls itself, thus forming a "nested" structure.

This concept is most easily explained through the use of an example. But first it is necessary to present the concept of the *external variable storage class*. An external variable is a global variable whose lifetime begins at declaration and extends throughout the program. (This is the sort of variable that is most commonly used in languages.) Hence one function can manipulate the value of an external variable, and any other function will have direct access to the same variable and this value (at least until it is changed). This class of variables is initially declared outside of all functions where it is generally *not* preceded with the keyword **extern**.

Now consider program 6.6 as presented in figure 6.8. Variable **a** is declared an external integer and initialized to zero on line 1. The rest of the program only consists of the function main(), which is comprised of four statements. During execution the first two statements are processed normally. First a new line and **Hello0** are output and then the external variable **a** is

```
          /* Program 6.6 */
/* An example of a recursive function call */
int a = 0;    /* external variable */

main()                          /* 5 */

{   printf("\nHello%d\t",a);
    ++a;
    if (a < 3)
        main();                 /* 10 */
    printf("Goodbye%d\t",a);
}
```

```
Hello0
Hello1
Hello2  Goodbye3   Goodbye3   Goodbye3
```

FIGURE 6.8. An example of a recursive construction. Here main() is a second order recursive function.

incremented. Next the condition expression on line 9 evaluates to true so the object statement on line 10 is executed. Since this substatement is a call to main(), control is passed to the first line in main(). Again a greeting is output with only the suffix differing. The second statement of the called main() again increments a, this time resulting in a value of 2. The condition again evaluates to true, and another call to itself is accordingly executed. The condition fails after the next greeting/incrementation cycle. Up to this point, the trailing statement, which outputs a suffixed farewell message, has not been executed. Because the third cycle does not trip the if statement, control passes to line 11 and finally the farewell message is output. Control then passes to the closing brace which acts as a return, sending control back to the calling function. Here the calling

function happens to be the second incarnation of main().
Control procedes from the function call on line 10 to line 11,
where once more the **Goodbye3** message is output. Afterwards
the closing body brace returns control to the original incarna-
tion of main() where the final message is output.

Recursion involves the nesting of the same function rou-
tine. The original function only terminates execution after all
of its later incarnations are executed. In fact, the last incar-
nation is the second function to finish execution etc.

Figure 6.9 presents a program that is in many respects
analogous to program 6.6. However recursion has the same
advantages as the iterative approach over the comparable
"brute force" control flow approach:

- Recursion can often conserve considerable coding
  time.
- Programs implementing recursion usually have
  a shorter source file length.
- A recursive approach allows a variable number of
  nesting levels.

Indirect recursion occurs when function a() calls another
function, which in turn calls another, until eventually a() is
called again: a() ⇨ b() ⇨ ... ⇨ a(). Because of the special
implications recursion presents to compiler design, it is not
allowed in some languages.

In program 6.6 it was essential that variable a have a
global lifespan as attained through the external class declara-
tion. Otherwise when a new incarnation of main() appeared,
the old value of a would be lost. Indeed if this variable was
automatic and initialized within the body of main(), each call
to main() would not ony create a new variable a but each
would be initialized to zero. An infinite "loop" would then be
set up since (a=1)<3 would always evaluate to true.

```
int a = 0;

/*      Program 6.7 - similar to pro-
  gram 6.6 except utilizes nested if's. with
  comments identifying corresponding parts */
main()
 /* original main() start */
(   printf("\nHello%d\t",a);
    ++a;
    if (a < 3)
     /* 2nd main() incarnation start */
       ( printf("\nHello%d\t",a);
         ++a;
         if (a < 3)
         /* 3rd main() incarn. start */
            ( printf("\nHello%d\t",a);
              ++a;
              printf("Goodbye%d\t",a);
            }
         /* 3rd incarn. end */
         printf("Goodbye%d\t",a);
       }
    /* 2nd incarn. end */
    printf("Goodbye%d\t",a);
 }
 /* original main() end */
```

Figure 6.9.   A nested if construction analogous to the recursive
program presented in figure 6.8.

The void Type-specifier

C has evolved somewhat since its commercial inception.
One of the changes to C was the addition of the type-specifier
void available on some compilers. A function header name
preceded by this keyword instructs the compiler that this
function is not to return any value. For example, the hypotheti-

cal function banter() could be declared void through the following first header line:

void banter(*arg1***,***arg2***...)**

If a program subsequently attempts to use a function call to banter() in an expression that is to be evaluated, a compilation time error message will result. For example, the expression:

sum = tot_1 + tot_2 + banter (*arg1***,...);**

would cause such an error message. On the other hand if a function is not explicitly declared void and no value is passed back through the use of a **return** (*expression*) statement, then a garbage value will be returned, and no error condition will be incited. The type-specifier void is a data validation device.

Formal Arguments in main() [Advanced]

It was mentioned in chapter 2 that main() rarely employs any formal arguments because no previous function is alive to pass information to it. Actually two occasions arise where formal arguments are present in main(): as command-line arguments and/or as program reset arguments.

A *command line* is the user-coded string given to the operating system to call a program to execution. In most systems, this is accomplished by typing in the name of the file holding the executable version of the program without the file name extension. For example on this book's reference system, the hypothetical executable file PROG1.EXE is invoked by keying PROG1 .

Some programs, however, are set up to require additional information after the file name (e.g. such as the name of associated data files, specifications on program versioning, etc.). Under many operating systems that support C, this additional information is passed to C as two argument values.

To illustrate this point, suppose that the following command line was input:

PROGX 2 , DATA4A

The additional command line information would be passed to main() as two formal argument values. The first value is the number of command line arguments the program was invoked with, or the *argument count*. In a command line, arguments are separated by one or more blanks. In the example there are four command line arguments: the executable file name PROGX, the numeral 2, the comma, and a utilized data file name.

The second value that is passed to main() is the pointer to an array of character strings that contain the actual command line arguments, with one argument per string. This pointer value is known as the command line *argument vector*. Consequently the header of main() in PROGX.C would appear as:

```
main(argc,argv)
int argc;
char*argv[];
```

By convention, the identifiers **argc** and **argv** are used to represent the command line argument count and argument vector respectively. Each formal argument is declared properly immediately after the function identifier line.

The programmer can reference the information conventionally by utilizing the character array **argv[]**. The array elements begin, as always, at subscript zero (i.e. **argv [0]** is PROGX) and progress up to the argcth element (**argv [argc]** or **argv [3]** is DATA4A).

Alternately, the pointer ***argv** can be used to indirectly reference the elements. This pointer variable is always initialized to the first element in the argument array. Incrementing the pointer by the operation **++argv** will cause the variable to point to the next element, which is the next command line

argument. The bounds of relevant information extends from
*argv to * (argv + argc - 1).

As demonstrated in the section on recursion, it is permis-
sable to call main(). As one would then expect, it is also
permissible to pass values to main(). Consider Program 6.8.
This is an example of direct recursion where main() is passed
values. While an arbitrary number of formal arguments can be
used in main(), the reader should remember that the first two
will initially contain the appropriate command line informa-
tion. This fact was used in program 6.8 to output a message
indicative of command line format (lines 9 and 11). The exter-
nal variable f_fl was used to trip if conditionals upon initial
execution. Argument variables b and c are changed in other
statements. Finally on line 22, the new values are passed to a
second incarnation of main(). Resetting of this sort has a very
limited area of applications; it can be used in conjunction with
a printf()/scanf() pair (s) to correct improper command line
strings without necessitating program termination.

```
/*        Program 6.8 - demonstrates the use of for-
   mal arguments in main() both as command line and
   program reset devices.  atoi() is a C library
   function that converts a string into its long
   int equivalent.                           */

int cla_fl =  0;        /* 7 */

   /* main() requires the following command line:
           prog68  int1  int2
     where int1 >= 0 and < 10, int2 = 1, 0 , or -1 */

main(arg1, arg2, arg3, arg4)
int arg1, arg3, arg4;
char *arg2[];           /* 15 */
```

```
{  /* declarations */
   if (cla_fl == 0)
      { arg3 = atoi(&arg2[1]);
        arg4 = atoi(&arg2[2]);
      }                     /* 21 */

   if ((arg1 != 3) || (arg3 < 0  || arg3 > 9) ||
       (arg4 != 0 && arg4 != 1 && arg4 != -1))
      { printf("\nYou have incorrectly typed the");
        printf(" command line.\nPlease input ");
        printf("the following values:");
        printf("\n\ninteger between 0 and 9:    ");
        scanf("%d", &arg3);            /* 29 */
        printf("\nflag value of -1, 0, or 1:  ");
        scanf("%d",&arg4);
        cla_fl = 1;
        main(3, 0, arg3, arg4);
      }
   else                      /* 35 */
      {  ; /* normal program statements */
         printf("\nCorrect command line ");
      }
}
```

```
PROG68 3 0 ]

You have incorrectly typed the command line.
Please input the following values:

integer between 0 and 9:   3

switch value of -1, 0, or 1:  0

Correct command line
```

Figure 6.10.   Recursive program demonstrating command line argument count use and alternate use of arguments.

7

Storage Classes

Introduction

Chapter 3 outlined the fundamental data types of C. Data types categorize a variable by the length of the memory block allocated to each variable and by the form in which these values are to be represented in this binary string. While the four data types are standard in C, the storage details are system dependent.

All variables also have another attendant attribute —

storage class. The storage class of a variable determines where in the program it can be used, how long the variable can be expected to store relevant values, and to some extent, how efficiently the CPU is utilized. There are four storage classes in C: automatic (**auto**), external (**extern**), **static**, and **register**. With the exception of **register**, no conflict arises between data type and storage class assignment. That is, aside from the above exception, any fundamental data type can be associated with any of the storage classes.

The two basic concepts underlying storage class differentiation are those of *longevity* and *scope* (or *globality*). The scope of a variable determines over what section(s) of the program a variable is actually "active" (i.e. may be referenced). For example in the last chapter, it was shown that automatic variables are private to (or local to) the function in which they are declared. If a statement attempts to utilize a variable that is in fact not local to that section, a compile time error message such as "UNDEFINED IDENTIFIER" will result.

Longevity deals with how long a variable actually continuously exits, in that it will faithfully retain a given value during execution of a program. Although a variable can be created, subsequently allowed to expire, then created anew, the reincarnation will not retain the original's value. So longevity also has a direct effect on the utility of a given variable.

The storage classes may also be broadly categorized as being either *internal* or *external.* Internal variables are declared and defined within a particular function, and their longevity and scope are closely associated with that function. The **auto, register**, and at times, **static** storage classes are internal.

External storage class variables are declared and defined outside of any function. This location does not imply that external variables are to be used by external statements (since external declarations are the only legally usable external statement type). Rather external variables are usable by one or

more functions, and indeed often by the whole program. The external category is comprised mainly of the **extern** storage class, but it also sometimes includes the **static** class. Note that the **static** storage class can have either an internal or external flavor.

The Automatic Storage Class

The notion of automatic variables was introduced in the last chapter. Automatic variables are declared at the beginning of the block in which they are to be utilized. Although the keyword **auto** may lead off a declaration as in:
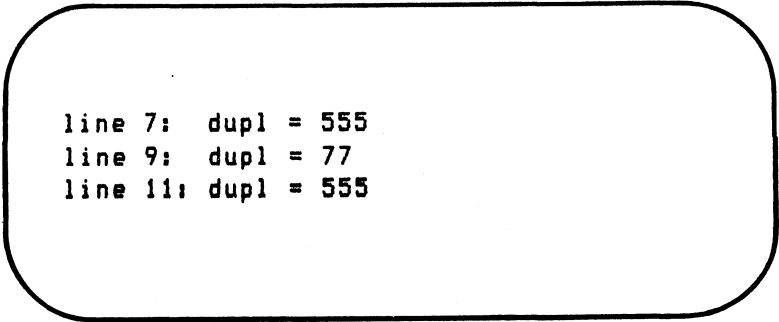
auto int i = 0, sum = 100, n;

this keyword is generally excluded, as declarations conducted within the function body are automatic by default.

The scope of an **auto** variable begins at its declaration and normally extends to the end of the block in which it was declared. However, if a variable with the same identifier is declared within a subblock of this region, the scope of the original variable does not include that subblock. This point is illustrated in program 7.1 in figure 7.1.

Program 7.1's main() contains two variables with the same identifier, **dupl**. The first is declared an **int** and is initialized to 555 in the first line of the body, line 6. Normally the scope of this variable would extend throughout the rest of main(). But in this case, a separate variable with an identical identifier was declared in the subblock starting on line 8. To resolve this duplicity, priority is granted to the second variable. When this inner subblock ends, however, the second variable's scope ends and the first variable is again dominant. The output of program 7.1 supports the previous discussion.

```
            /* Program 7.1 */
/* duplication of an internal (auto)
   identifier within a subblock        */
main()
                                  /* 5 */
{  auto int dupl = 555;      /* auto opt */
   printf("\nline 7:   dupl = %d",dupl);
   { char dupl = 77;
     printf("\nline 9:   dupl = %d",dupl);
   }                         /* 10 */
   printf("\nline 11: dupl = %d\n",dupl);
}
```

```
line 7:   dupl = 555
line 9:   dupl = 77
line 11: dupl = 555
```

Figure 7.1.   Program illustrating the exception to the
                  scope of internal variables.

Just as scope rules determine where a variable is active, longevity rules determine over what period a variable is still "alive". For an automatic variable, the life begins with its declaration and extends to the end of the block in which it was declared. This is similar to the corresponding scope rule except there is no exception for identifier duplicity. As a result, although the original dupl in program 7.1 was not active within the inner block of lines 8 thru 10, nonetheless it was alive. And while the original variable could not be referenced in

this subblock, it retained its correct value, as line 11's output verifies.

Identifiers local to entirely different blocks or functions can, of course, utilize the same name without conflict since their scope does not overlap. For example it was shown in the last chapter that formal and actual matching arguments may bear identical names without compromising uniqueness. Variables declared in the same block must have different significant characters in their identifiers or a compile time error condition results.

When the life of a variable ceases, its memory block is freed for other purposes. Without the memory block, a variable, in fact, ceases to exist (i.e. "it dies"). Naturally no value can then be held by it. Refer to program 7.2 presented in figure 7.2. This program is comprised of two functions — main() and cnd_out(). The latter function assigns the value passed to its first argument, **output**, into the variable **copy** if and only if the value passed to its second argument is **y**. This process occurs when cnd_out() is called from main() on line 6. The output truthfully reflects the passed value of 666.
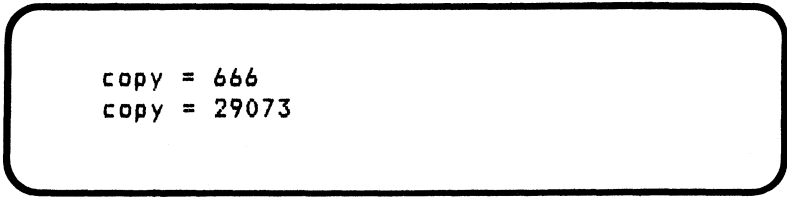
In contrast, if the second passed value is not **y**, then the first passed value is not assigned to **copy**, yet the value of this variable is still output. This situation occurs when the call in line 8 is executed. The point that is of special interest is the fact that **copy** in the second call did not retain its previous value of 666. When the first function call was completed, the control passed from the parent block to which copy belonged (i.e. cnd_out()'s body). At this point, the variable **copy** ceased to exist. Subsequently the second call to cnd_out() initiated the recreation of **copy** (as well as **output** and **dcn_fl**). The second output merely reflected the contents of the memory block allocated to the second incarnation of **copy**. This situation rarely occurs because the automatic variables of auxiliary functions are typically assigned values through argument matching, initialization, or through expression assignment.

```
                /* Program 7.2 */
/* illustrates longevity of auto sc */
main()

{   char o_fl = 'y';      /* 5 */
    cnd_out(666, o_fl);
    o_fl = 'n';
    cnd_out(777, o_fl);
}
                              /* 10 */
cnd_out(output, dcn_fl)
int output, dcn_fl;

{   int copy;
    if (dcn_fl == 'y')   /* 15 */
       copy = output;
    printf("\n copy = %d",copy);
}
```

```
copy = 666
copy = 29073
```

Figure 7.2.   Program illustrating the longevity of
automatic variables.

Formal arguments are also automatic, although they are not declared within a block, but rather in the function header. Their longevity and normally their scope extend throughout the ensuing function. Just as other automatic variables, formal arguments do not mechanically retain their values between parent function invocations.

There are two consequences of the scope and longevity rules of auto's worthy of note at this point. First, any variable local to (almost) all of main() will normally be alive through-

out the whole program although such a variable is only active in main(). This results from the fact that a program begins with main() and normally terminates with the closing brace of main(), which is the definition of the enclosing block of a variable local to main(). Secondly, during recursion, the nested variables are not actually reincarnations of the original (as described in the last chapter), but unique auto variables. This situation is similar to block-nested auto variables with identical names.

EXCEPTION TO LONGEVITY RULE [ADVANCED]

The one exception to the longevity rule for auto variables involves the use of goto's. Whenever a goto directs control into a subblock, the variables local to that block become active even though the goto may have bypassed block declarations.

Examine figure 7.3. Program 7.3 contains two variables with the identifier **value**. Accordingly scope and longevity rules for these variables should follow those explained for **dupl** in program 7.1. Notice that the goto statement of line 18 sends control back into the subblock, but supercedes the declaration of line 8 and incrementation statement of line 9. During the first and only full execution of this block, the goto bypass does not apply as execution falls through from line 7. Afterwards the conditional goto directs only partial execution of this block starting after the declarator and incrementation of **value**. Furthermore the goto is encountered after the close of the subblock so it should be expected that the second or inner **value** ceased to exist when this goto is executed.

The output clearly proves that this assumption is incorrect. The values output by the inner block's printf() are those of the inner variable **value** for all three executions. Lines 8 and 9 have obviously been bypassed on the second and third loops because **value** is only increased by two on line 14 and not by three as if both lines 14 and 4 had been executed. Conse-

```
                /* Program 7.3 */
    /* illustrates longevity and scope of */
      /* auto sc variables with goto's */
    main()
                               /* 5 */
    {   int cnt = 0;
        int value = 0;
        {   int value = 68;
            ++value;
      loop:       ;              /* 10 */
            ++cnt;
            printf("\ntrial #%d",cnt);
            printf("\tvalue = %d",value);
            value += 2;
        }                        /* 15 */
        --value;
        if (cnt < 3)
            goto loop;
        printf("\n\n\"original\" value = ");
        printf("%d\n",value);    /* 20 */
    }
```

```
trial #1        value = 69
trial #2        value = 71
trial #3        value = 73

"original" value = -3
```

Figure 7.3. An example of implicit local variable declaration when a goto is used to enter a subblock.

quently, all local variables are implicitly declared when a goto directs control into their subblock.

Not only is the inner **value** active during block execution but it also retains its value between executions despite the subblock termination. This increased longevity is not guaranteed in C, but occurs in most systems because of compiler design.

The register Storage Class

There are three categories of memory available in most computers: secondary, main, and register storage. Secondary storage is usually of a magnetic nature, such as a floppy or hard disks, or magnetic tape. This kind of storage medium cannot generally be used directly by a computer. Generally data and programs must be loaded first into main memory before the CPU can address this information.

Main memory is usually comprised of semiconductor components (RAM or ROM). Main memory represents the bulk of the memory capacity available to a computer for program execution. Nonetheless there are a few specially designed registers in the CPU that are essential for computer operation because all actual manipulations involve these registers. (These registers are always of a semiconductor nature, as is the entire CPU.)

When a variable's value is needed during program execution, there are normally a number of steps that must be taken to deliver this value to the CPU:

1. The memory address of the variable is loaded into a register of the CPU.
2. The CPU sends this address to memory where the memory management unit locates the address and returns a copy of the contents of the referenced block.
3. The value of the referenced block is stored in one of the registers of the CPU.

These steps assume that the value desired is not presently in one of the registers of the CPU, as would be the case if this value had recently been already used in the CPU and still resided in one of the registers. For example given the source code section,

```
tot = 35 * SUM + rsd;
gtot = tot * intrt;
```

it would be unnecessary to retrieve the value of **tot** from main memory for the second statement because that value should still reside in one of the registers from the execution of the previous step.

Decisions such as how to best handle CPU usage and when to store values in memory are not generally the concern of higher level language programmers. Yet sometimes there are instances where a great execution time savings could be realized if a value was consistently stored in a register. For example, if a simple loop needed to be executed a thousand times, it would save a considerable amount of time if the loop counter and perhaps a few other variables were stored continuously in registers.

C includes a special storage class that is much like auto except that it strongly suggests to the compiler that the associated variable be kept in a register. This is the **register** storage class.*

The register keyword can only be applied to automatic variables, and depending on the machine only certain data types may be used as register variables (char, short int, and int are typically allowed). Because of physical as well as software limitations, only a few variable values may be stored in registers. So depending on the architecture of the machine

---

* Some less expensive compilers do not support this storage class. Often all register declarations are converted to auto.

(specifically how many registers it has) and the number of previously designated **register** declarations, a **register** declaration request to the compiler may fail.* When this occurs, the compiler treats all disqualified **register** variables as **auto**'s.

The **register** declaration takes the following form:

register *data-type identifier initialization*opt;

as demonstrated by the following two declarations:

```
register int i = 0;
register short int cnt;
```

The **int** keyword can be omitted as it is the default declarator. Formal arguments can also be declared **register** at the proper location (i.e. between the line with the function identification and the opening brace of the same function).

register variables follow the same scope and longevity rules as **auto**'s. Their declaration, like any internal variables must occur at the beginning of a block. The only limitation connected with this storage class is that the "address of" operator, **&**, may not be applied to a **register** variable.

The External (extern) Storage Class

In languages such as BASIC, all variables are both alive and active throughout the entire program. Variables of this kind are known as external, common, or global variables. External variables are declared outside of a function, most commonly near the beginning of a source file. For example, the external declaration of integer **k** and double precision **x** and **y** might appear as:

---

* Mini and microcomputers can generally only handle a few such declarations.

```
int k = 1;
double x, y = .271e-5;

main()

{...
}
```

The keyword **extern** could optionally begin each declaration. However such an inclusion does confuse some compilers as **extern** is more appropriately used for external redeclaration, as discussed next.

Once a variable has been declared external, it may be used directly by any of the subsequent functions in the same source file. If one function changes the value of an external variable, then subsequent functions can reference only that new value.

There are two circumstances where it is necessary to redeclare an external variable within the body of a function:

- When the function to use an external variable is defined prior to the declaration of that variable (i.e. the function is located above the declaration in the same source file).
- When the using function is located within an entirely different source file.

A redeclaration must occur in each and every function that conforms to either of the above descriptions. *These redeclarations must begin with the keyword* **extern**. For example, suppose that the hypothetical function in_file2() needed to reference the two external variables **k** and **y**, originally declared and defined in a different file. The resulting source file would appear as:

```
          /* File 1 */
int k = 1;
double x, y = .211e-5;
  .    .    .
```

```
     /* File 2 */
 .   .   .
type-specifier in_file(formal argument list)
formal argument declarations
{   extern int k;
{   extern double y; /* x not utilized */
    .  .  .
}
 .   .   .
```

extern redeclarations appear, by convention, after called function declarations but before automatic variable declarations. Notice that the redeclarations do not include the initialized values. This is due to the fact that local redeclarations of **extern** variables do not serve to define the variable, but only to notify the compiler of a non-standard identifier usage. In this regard, external redeclarations are analogous to those necessary for called functions returning non-integers.*

The nature of external variables is demonstated in program 7.4, contained in figure 7.4. This program is composed of functions from two files, FILEA and FILEB. Their divisions are delineated by the comments on lines 5a and 1b.

There are two **extern** variables in this program — **tot** and **RATIO**. The latter is capitalized by convention because it is a symbolic constant. External variables are sometimes used in this manner to provide constant values to an entire program.

Notice that the external variable declarations are quite identifiable, even without the keyword **extern**, because of their position outside of any function. Because these external declarations were, by convention, placed at the beginning of a file before any function definitions, there was no immediate need to redeclare them within the functions of this same file.

---

* However for externals, the redeclaration mainly informs the compiler where to look for and how to handle the variable, while called function redeclarations allow the compiler to transfer non-integer returned values correctly.

```
        /* Program 7.4 */
/* illustrates nature of extern variables,
        divided into two files */

    /* FILE A */          /* 5a */
int tot = 0;
double RATIO = 3.27;

/* arbitrary tot manipulation */
main()              /* 10a */
{   double abs();
    int ext_fact();
    int a = 1, b = 2, c = -17;
    tot = a + b + c;
    if (tot != 0)              /* 15a */
        tot = abs((double)tot);
    printf("\n tot = %d",tot);
    tot = tot / RATIO;
    printf("\n tot = %d",tot);
    ext_fact();                /* 20a */
    printf("\n tot = %d",tot);
}

/* returns absolute value of double */
double abs(x)              /* 25a */
double x;

{   x = (x < 0) ? (-x) : x ;
    return (x);
}                          /* 30a */


    /* FILE B */
/* directly assigns extern tot  */
    /* the factorial of tot */
ext_fact()
                           /* 5b */
{   extern tot;
    int i;
    for (i = tot - 1; i > 0; --i)
        tot *= i;
}                          /* 10b */
```

```
tot = 14
tot = 4
tot = 24
```

Figure 7.4. Program demonstrating the use of external variables.

Program 7.4 demonstates some common ways in which external variables can be used. In line 14a, **tot** is reassigned a value through an operational statement. Within a function, external variables can be utilized as any automatic variable would be. On line 18a, **tot** is again reassigned a value in an operational statement, but this time all the operands are external variables. Even though the latter statement references only externals, it must be located within a function body. C allows only external declarations, comments, and macroprocessor "statements" (chapter 8) to lie outside the bounds of a function.

In line 15a, the external **tot** is used in a condition expresion. In the next line, **tot** is again assigned a new value; however the rvalue expression involves a function call that utilizes **tot** as an actual argument. The cast operator was applied to **tot** because the called function, **abs()** demands a double precision floating point passed value. A copy of the value of **tot** is assigned to the actual argument **x** in **abs()**, just as would occur if **tot** was automatic. The value returned by **abs()** is then used to reassign a new value to **tot**.

In contrast, the function call of line 20a neither passes nor returns any values, but rather relies on the globality of **tot** to convey data. In **tot_fact()**, the variable **tot** is manipulated directly. After control passes from **tot_fact()**, the value of **tot**

reflects whatever reassignments, if any, it underwent within the aforementioned function. *Consequently, external variables can violate the privacy of functions.*

It is very important to note that in the header of tot_-fact(), **tot** was not declared as a formal argument. If **tot** had been erroneously declared in such a manner, as in the lines:

```
tot_fact()
int tot;
{ ...
```

then an automatic variable with an identifier of **tot** would have been created within tot_fact(). Since there are no actual or formal arguments within the parentheses following the identifier **tot_fact** on lines 20a and 4a respectively, no value would have been passed to the newly created automatic variable **tot**. It would, therefore, be initialized to garbage.

Note that because tot_fact() is located within another file, the variable **tot** which it utilizes from file A, must be locally redeclared using the keyword extern. This was done on line 6b. Failure to include this line would have resulted in a compile time error message.

Obviously program 7.4 has little practical value. The statements in main() are arbitrary aside from what they reveal of the nature of the extern storage class. Also, logically the function tot_fact() would probably have been part of file A since it is closely associated with external **tot** of that file.

The static Storage Class

The **static** storage class can have either an external or an internal flavor. In either case variables of this class must be declared using the keyword static. As the name suggests, static variables have a longer lifespan than do their auto or register counterparts. In fact static longevity equals that for

extern's. It begins upon declaration and continues until the end of the program. The scope of static's differ depending on whether they are external or internal.

INTERNAL static VARIABLES

An internal static variable is declared at the beginning of the block in which it is to be used. The scope of the variable extends throughout only that block. Therefore internal static variables are similar to automatic variables, except that they remain in existence throughout the remainder of the program, whereas auto's are recreated with every function invocation. Accordingly, internal static variables can be used to retain values between function calls without compromising function privacy as extern's do.

An example of the implementation of the static storage class is presented by program 7.5 in figure 7.5. In this program main() is designed as a driver to function instr_st(). The latter function accepts one of three commands (i.e. passed values). A command of s will set or reset the value of the static variable cnt to zero. A command of c will increment this variable by one. p causes a message and the value of cnt to be output. Note that the default data type of int has been used in the declaration of cnt on line 15.

The first statement of interest in main() is line 6. This statement is a function call to intr_st() that assigns a value of zero to cnt. The next statement is a classically controlled for loop that iterates nine times (i.e. from -4 to 4 inclusive, in increments of 1). During each repetition, its object statement executes a call that increments cnt by one on each call. Finally line 9 sends the command to execute the normal output message (lines 21 and 22).

If cnt had been declared an automatic variable, then cnt would cease to exist between calls to intr_st(). As a result, each new incarnation of cnt would be initialized to the garbage value

```
/*          Program 7.5 - shows the
   utility of internal static variables */
main()

{  int i;            /* 5 */
   intr_st('s');
   for (i = -4; i < 5; ++i)
       intr_st('c');
   intr_st('p');
}                    /* 10 */

intr_st(com)
char com;

{  static cnt;       /* 15 */
   if (com == 's')
       cnt = 0;
   else if (com == 'c')
       ++cnt;
   else if (com == 'p')     /* 20 */
       { printf("\nThe # of count comm");
         printf("ands is %d\n",cnt);
       }
   else                     /* 24 */
       { printf("\n\tWrong command on");
         printf(" invocatoin #%d",cnt);
       }
}
```

```
The # of count commands is 9
```

Figure 7.5. Program demonstrating the longevity aspects
of the internal static storage class.

that happened to be in the memory block allocated to it.

There is normally no point in using internal **static** variables within **main()**. This is due to the fact that an **auto** in **main()** has program long longevity because **main()** is the program driver. There are two rare instances when internal **static**'s may be helpful in **main()**.

- They can be declared within subblocks that are encompassed by a goto / label construction to ensure value retention between iternations.
- They can be useful in those rare instances where a program is designed with calls to the function main().

EXTERNAL static VARIABLES

External **static** variables are declared outside of all functions, much as **extern** variables are. However, external static variables must include an initial **static** keyword. For example, the declaration of an **extern double** variable **e** and **static** integer **s**, both initialized to one, would appear above **main()** as:

```
double e = 1.0;
static int s = 1;

main()
{ ...
```

External **static** variables differ from **extern**'s in that their scope is limited to the remainder of the file in which they are declared. If a function from one file attempts to reference another file's external **static** variable, a compile time error will result. So in this manner, external **static**'s provide file security and privacy not inherent to **extern**'s.

Declaration and Initialization

Although the formal rules for declaration of (extended) fundamental data type variables have been presented piece-meal throughout the preceding chapters, these rules are presented here for review. For the declarator:

- The indicative storage class keyword must begin the declaration statement. Then after at least one blank space, the optional data type extension (i.e. short, long, and/or unsigned), if used, must appear. Lastly, after at least one blank, the fundamental data type must be supplied.

The two exceptions to this rule make use of the default rules of C, and they are:

- The keyword extern may be omitted in external delarations if either the data type or extension specifier, or both are present. Likewise the keyword auto may be omitted for internal declarations.
- The keyword int may be omitted from internal or external declarations if the data class or extension specifier or both are present.

Therefore if the following declaration:

short x;

was encountered outside of all function bounds, it would declare the variable x as a **extern short int**. The keyword **extern** may never be used to declare external **static** variables.

Variables can be explicitly initialized to the following values:

- A constant expression as discussed in chapter 4.
- The address of a previously declared external or static variable, through the use of the "address of" operator, **&**, perhaps offset by a constant value.
- The address of a function optionally offset by a constant value.

- The value of a previously defined variable or function (for *auto and register variables only*).
- An expression involving one or more of the above elements.

Generally it is wise to be conservative with initializations. The use of a separate assignment statement avoids the restrictions inherent on initializers.

Automatic and **register** variables that are not explicitly initialized are initialized to whatever value happens to reside in their allocated memory block. In contrast, *static and extern's are implicitly initialized to zero by default*. This is an especially convenient attribute in internal **static** variables, as set conditional statements (e.g. line 16 and 17 in program 7.5) can be avoided.

Storage Classes and Recursion [Advanced]

Recursion is both a nested and an iterative process. Like normal loop constructions, there must be a mechanism present to terminate the recursion process or an infinite chain will be set up. This section deals with what storage class of variable should be used as such a limiting mechanism.

It has been shown that a function's automatic variables are created anew for each invocation of that function. In the special case of recursion, there exists a unique, identically named set of automatic variables at every level of the recursion process. Furthermore because these recursively-nested **auto** variables have duplicate identifiers, their scopes are limited only to the level at which they were defined. This attribute obviously leaves **auto** variables unsuited as the mechanism to control recursion.

Parameters can be used to pass information to the nested reincarnations of the original function via argument matching. This scheme has two drawbacks, however:

- Only one value may be returned back to the parent function.
- The process of value passing is relatively slow and cumbersome.

Additionally for indirect recursion, passed values may be required to "trail along" through those functions that do not need them.

External flavored variables have none of above-mentioned flaws, yet it is usually wise not to use externals when there is an equally good substitute. Fortunately in this instance there is just such a storage class — namely internal **static**. Unlike their **auto** counterparts, internal **static** variables are not created anew between function invocations, including recursive ones. There are none of the inefficiencies associated with value passing because **static**'s have "program wide" longevity. Lastly, because they are a part of one function they do not present any problems associated with the misuse of external storage classes.

Consider the directly recursive function **recursv()** in program 7.6, presented in figure 7.6. This function calls itself from line 23. This line is one of the component simple statements comprising the compound object statement of the **if** statement of lines 17 through 25. Consequently the condition expression of the **if**, on lines 17 and 18, determines to what depth the recursion occurs. Inspection of this program plainly shows that the recursive process is classically controlled. However, there are three rather than the normal single, loop counters.

These three counters are of different storage classes: **extern**, internal **static**, and "**auto**" (formal parameter). All three variables are assigned an initial value of zero — the first two by the default mechanism and the last through a passed value from **main()** (line 7). In **recursv()** there are separate lines to increment each variable, but these occur in very different places. The parameter incrementation occurs on 16, before the

```
                /* Program 7.6 */
/* demonstrates the use of different storage
   class variables as recursion counters   */

int extrnl;            /* 5 */
main()
{  recursv(0);
   printf("\nFinal value of extrnl ");
   printf("= %d",extrnl);
}                      /* 10 */


recursv(par)
int par;

{  static intr_st;    /* 15 */
   ++par;
   if  ( (intr_st < 4) && (par < 4)
         && (extrnl < 4) )
       { ++intr_st;          /* 19 */
         printf("\n par = %d",par);
         printf("\t intr_st = %d", intr_st);
         printf("\t extrnl = %d", extrnl);
         recursv(par);
         ++extrnl;
       }               /* 25 */
}
```

```
  par = 1          intr_st = 1      extrnl = 0
  par = 2          intr_st = 2      extrnl = 0
  par = 3          intr_st = 3      extrnl = 0
Final value of extrnl = 3
```

Figure 7.6. Program illustrating the use of the extern and
internal static storage classes and a parameter to
classically control a direct recursion chain. (The
external is used incorrectly.)

if statement. The internal **static** and external variables are incremented from within the **if**, however the former operation (line 19) occurs before the recursive function call of line 23, while the latter operation occurs immediately after (line 24). The location of each incrementation statement will be shortly shown to be very important.

The values of the **static** and external counters can be referenced directly by each recursive incarnation. This is not true for the parameter because it follows the rules for **auto** storage class. Therefore in each reincarnation a unique and separate variable **par** will occur. The value of each **par** must be passed in turn to the next recursively nested variable **par**. This is accomplished by argument passing. The recursive function call on line 23 passes a copy of the actual argument **par** to the formal argument **par** of the recursively reincarnated function. Without this argument matching the called function's **par** would be initialized to garbage.

Now consider the overall execution of program 7.6. The first line of **main()** calls **recursv()** and matches a value of zero onto the latter function's parameter **par**. The first statement in **recursv()** is the declaration and default initialization of the internal **static intr_st**. Then in line 16 **par** is incremented (to one). The test condition in lines 17 and 18 evaluates to true as **intr_st == 0, par == 1,** and **extrnl == 0**. Therefore the object block begins execution with the incrementation of **intr_st** (to one). Then after three self-evident **printf()**'s, the recursive call is encountered on line 23, so control passes to the first "reincarnation" of **recursv().*** The call passes the value of its actual argument (i.e. **par == 1**) to the formal argument **par** of the reincarnated function.

---

* Unlike automatic variables and parameters, the code for a function is actually reused during recursion. It is, however, conceptually useful to visualize this process as a creation of a totally new, duplicate function.

Again the first line encountered is the declaration of the internal static **intr_st**. This declaration does not create a duplicate variable because static's remain alive between function invocations. Note also that this time the variable is not default initialized to zero. *Default initialization to zero only occurs during the creation (definition) of a extern or static variable.*

The next statement increments the new duplicate parameter to two. The test condition in lines 17 and 18 again evaluates to true with **intr_st** == 1, **par** ==2, and **extrnl** == 0. Consequently the compound object block (up to the recursive call on line 23) is executed almost as before, only the specific values of the variables differ.

The second reincarnation of **recursv()** is a playback of the first reincarnation, with only specific variable values differing. It is in the third reincarnation that the test conditions fails after **par** is incremented to four. The close of the function is encountered after the subblock so control passes back to after the point of the function call in the second reincarnation.

In other words, line 24 is finally executed, incrementing **extrnl** to one. Next the end of the second function incarnation is reached so control passes back to after the point of the function call in the first incarnation of **recursv()**. Again the external variable is incremented; then the closing brace of the function is encountered. This time however, control passes back to the original **recursv()**. Here **extrnl** is incremented to its final value of three; then control passes back to **main()** to line 8. Two **printf()**'s are then executed and the program terminates.

There are two main points that can be learned from program 7.6. The first concerns the placement of the statement that reassigns values to the limiting variables. As was the case with the incrementation of **extrnl**, the placement of such a statement after the recursive call is useless. Statements in this relative position are only executed on the "way back" from

recursive nesting. Care must also be taken as to where the reassignment statement appears in relationship to any conditional tests. For example, although the output shows both **par** and **intr_st** to have the same value during output routines, the value of **par** is actually one greater than **intr_st**'s during test condition evaluation. This is so because **par** is incremented before the if's test condition while **intr_st** is incremented afterwards.

Secondly the advantages of the internal **static** storage class when used in recursion should be evident. These advantages include:

- default initialization to zero
- recursion-wide longevity; no creation of duplicate short-lived variables.
- avoidance of the necessity of argument passing and returned values.

Despite these advantages, recursive parameter and returned value passing is often used in simple instances of direct recursion because of its conceptual clarity. And in a related manner, automatic variable and parameter usage minimizes the risk of accidental data transference between recursive function reincarnations.

The use of externals during recursion may be advantageous for cases of complicated indirect recursive processes. Their use can eliminate large argument lists, and the necessity of "just tagging along" argument values.

Functions and Privacy

Functions themselves are considered external entities. Functions cannot be found defined inside one another, and they are normally available throughout the entire program. However if a type-specifier of the function header is preceded with the keyword **static**, then the function's scope is limited to

the file in which it is located. This is completely analogous to external **static** variables. On these grounds functions are not typically private but rather their variables are. Chapter 9 will demonstrate that even internal variable privacy can be violated through the use of pointers.

Programmers new to the concept of local or internal variables often ignore them as well as what they consider the burden of argument and returned value passing by over-utilizing external variables. An approach of this sort quickly defeats some of the most alluring attributes of C — namely C's modularity and portability.

There are a number of areas where the use of external variables is suggested despite any possible tradeoff in design aesthetics. External declarations are often used:

- To avoid extremely long argument lists. This type of usage should be reserved for functions that are closely associated because calls using such a format are generally nonportable.
- As intermediate, "temporary explicits" to convey information when a function must "return" more than one value. External variables can be used as the repositories of a function's manipulations. The calling function can subsequently assign the external values to internal variables. However pointers (chapter 9) are usually more appropriate in these cases.
- When two functions must share data but are not directly connected through a function call. Rather than have data tagging along through intermediate functions, the data can be stored in an appropriate external variable.
- Because only external and static arrays can be initialized (chapter 9).
- When coding and execution efficiency outweigh all other considerations. Direct referencing of an external variable is naturally more efficient than the alternative argument matching scheme.

Program Level Hierarchy [Advanced]

The format of C programs gives rise to an observable structure whose elements are subblocks, blocks, functions, files, etc. The different levels of a program's structure also provide a hierarchy which resolves any dispute in variable uniqueness, thereby ensuring the integrity of a unit of code.

Refer to figure 7.7. This figure graphically defines the precedence assigned to the levels of a C program with regards to variable uniqueness. When the scope of two or more variables with the same identifier overlaps in a region, the variable defined within the level of highest priority (i.e. the inner level) is assigned jurisdiction (scope) over this region. The other lower priority duplicate variable cannot be directly referenced in this region.* This latter variable retains scope over whatever portion of its normal range remains after the higher priority region is deducted.

While the scope of a variable may be diminished due to the presence of an inner level duplicate variable, the longevity of both variables remain unchanged.

For example, it has been shown that the scope of an auto extends throughout the rest of the block in which it was declared *except* if there exists a subblock within this region that contains the declaration of an identically named variable. Then within that subblock, the newly declared variable reins. The longevity range of the outer, lower priority variable remains unchanged, but its scope no longer extends into the aforementioned subblock. Exceptions of this type can be noted throughout the level structure outlined in figure 7.7.

---

* A pointer variable could still be used to "indirectly" reference and manipulate the value of the lower priority variable even in this region. Aspects of pointers will be covered in chapter 9.

innermost subblock   HIGHEST PRIORITY
↓
next outer block
↓
. . .
↓
function body
↓
formal arguments (parameters)
↓
inclusive object file
↓
**other object files
(program)**   LOWEST PRIORITY
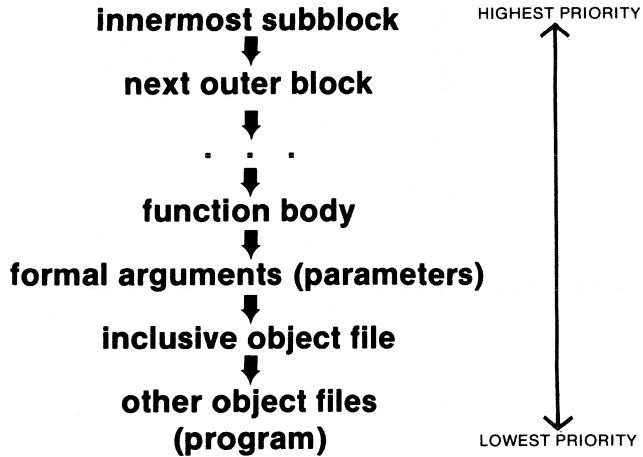
Figure 7.7.   Natural level hierarchy for resolving conflict in
identifier uniqueness.

The concept of variable uniqueness and privacy is thoroughly demonstrated in program 7.7, presented in figure 7.8. This program is divided into three functions in two files: main() and ext() reside in file A, while d_file() resides in file B.

In file A there are five variables with the identifier **x**: the extern double x declared on line 7a, the int x declared on line 10a, the static int x declared on line 12a the char x declared on line 14a, and the unsigned int declared on line 27a. The hierarchical program levels resolve all scope disputes between these variables as discussed next.

The hierarchical block scheme for duplicate internal variables is demonstrated to a level three deep. As demonstrated previously, the scope of an internal begins at declaration and continues to the end of that block unless a subblock introduces an identically named ("duplicate") variable. Then the previously described scope adjustments apply.* The first four

---

* Notice that program format, especially indentation and brace placement, plays a very important rule in this program. The hierarchical relationship of the duplicate internal variables in main() is immediately apparent due to proper format.

```
/*       Program 7.7 - illustrates vari-
  able uniqueness with regards to block,
  function, and file reorganization. Divided
  into 3 functions in 2 files.             */

            /* FILE A */
double x = 1.0;

main()
{  int x = 2;                    /* 10a */
   printf("\n\t   IN FILE A");
   {  static x = 3;
      printf("\nib1 x = %d",x);
      {  char x = 5;
         printf("\nib2 x = %d",x);
      }                          /* 16a */
      printf("\nib1 x = %d",x);
   }
   printf("\npb   x = %d",x);
   ext();                        /* 20a */
   d_file(x);
}

ext()
                                 /* 25a */
{  printf("\next x = %.21f",x);
   {  unsigned x = 7;
      printf("\nibc x = %u",x);
   }
}


            /* FILE B */
int x = 11;

d_file(z)
int z;              /* 5b */
```

```
{   printf("\n\t   IN FILE B");
    printf("\npar z = %d",z);
    {   long z = 13;
        printf("\nvar z = %ld",z);
    }                   /* 11b */
    printf("\next x = %d",x);
}
```

```
                IN FILE A
ib1 x = 3
ib2 x = 5
ib1 x = 3
pb   x = 2
ext x = 1.00
ibc x = 7
                IN FILE B
par z = 2
var z = 13
ext x = 11
C>
```

Figure 7.8. Program illustrating the effect program structure hierarchy has on resolving variable identifier duplication conflicts.

variable output printf()'s on lines 17a and 19a also establish that the longevity of internal variables continues unbroken throughout the block in which they were declared despite any occurrence of subblocks with duplicate variables.

The external variable **x** cannot be directly referenced in main() at all. This is due to the fact that there is a duplicate variable (i.e. **int x** = **2**) declared immediately after the opening brace of main(). Because this internal variable is declared at the level of the function body, all subsequent statements of this function are removed from the scope of the external **x**. Externals cannot be referenced in any block in which a duplicate

variable is declared.* In a similar way parameters that duplicate external variable names disallow use of said external throughout the entire function.

If a function level duplicate of an external variable does not exist, then of course the function may reference (and manipulate) the external variable. For example, ext()'s outer block references the extern x = 1.0. The inner block of this function cannot because a duplicate variable (i.e. **unsigned x=7**) is declared.

Control passes to d_file() through the function call on line 21a. This call passes a copy of the variable x, active on the outermost block of main() (i.e. **int x = 2 ;**), to d_file(). The called function subsequently matches this value to the formal argument z. Formal arguments obey the rules of auto's except their longevity always includes the entire called function by necessity of declaration position. Notice that the scope of the argument z does not include the subblock of lines 9b through 11b because the duplicate internal **long z = 13** is declared within this block.

File B also contains an external variable x that may be referenced directly by any function in this file. When two files containing external, non-static variables are linked together, a warning such as "multiple definition of *variable-identifier*" will be produced. Nevertheless the linking process will be successful (with most linkers). However each file will be unable to directly access the other's "duplicate" extern variable — in effect both now become external static in regards to those two files. Thus line 12b must output the value of x externally defined in its file.

---

* Actually the external variable can be referenced up to the point of the duplicate variable's declaration within the associated block. That is, previous declarations can make use of the external variable's value, but may not change the actual external's held value.

This system of precedence, although it has some implications for software security, was primarily instituted to resolve identifier conflicts. Normally, lengthy programs have a way of utilizing all the common variable names (esp. the short identifiers commonly used to hold integer values — i, k, l, m, and n). However by using subblock declarations, the same name may be reused for similar purposes without fear of contamination.

There are several drawbacks associated with the use of duplicate variables. As explained, there are two inherent limitations that result from the nature of program level hierarchy:

- The same identifier cannot be used to declare two different variables on the same level (and, if applicable, within the same block). An attempt to do so will generate a redefinition compile time error.
- Within any level it is only possible to directly reference and manipulate the duplicate variable whose scope includes that level. Similarly named duplicate variables residing on different levels cannot be directly referenced in the same level of code.

Perhaps even more critical than the above two restrictions is the possibility of degrading program clarity by overusing this facility. Impartial identifier reuse can result in confusing code. (This is the same admonition given to the reader concerning actual to formal argument identifier duplication.)

8

Compilation and the C Preprocessor

Introduction

Regardless of what computer language is being used, pro-
gram code must be translated into a form usable by the CPU
—namely machine code. For numeric and alphanumeric data,
this involves a fairly simple conversion into a binary represen-
tation (data types) as discussed in chapter 3. However the
conversion of statements from a higher level language like C

into machine code form is a very complicated process. It is not unusual for one high level statement to resolve to tens or even hundreds of machine level operations. Furthermore this binary, machine executable code is highly machine dependent. Executable code from one computer will rarely run properly on another model, although there are programs available to effect this transposition.

In compiled languages such as C, this translation must be completed only once before the program can be executed.* The end product is an executable file, which may be invoked by specifying the file name without the filename extension. However as the compilation process in C is generally more involved than with most other high level languages, it is mandatory that the C programmer understand the basics of compilation.

Program Implementation Steps

Figure 8.1 contains a type of flow chart, or *data flow diagram*, of the steps involved in coding, compiling, and executing a C program. The first step involves the use of any text editor system to create a source file containing a C language program. Text editors are generally included as part of an operating system package.

Next the source file is presented to the compiler program. C compilation can be divided into three sequential steps:

1. *preprocessor phase* — the compiler identifies and processes all preprocessor command "statements" (i.e. those beginning with a # in column one), producing "extended C source code".

---

* Generally there are no hard and fast rules as to which languages are interpreted and which are compiled. As a case in point, compiled versions of BASIC and interpreted versions of C are available.
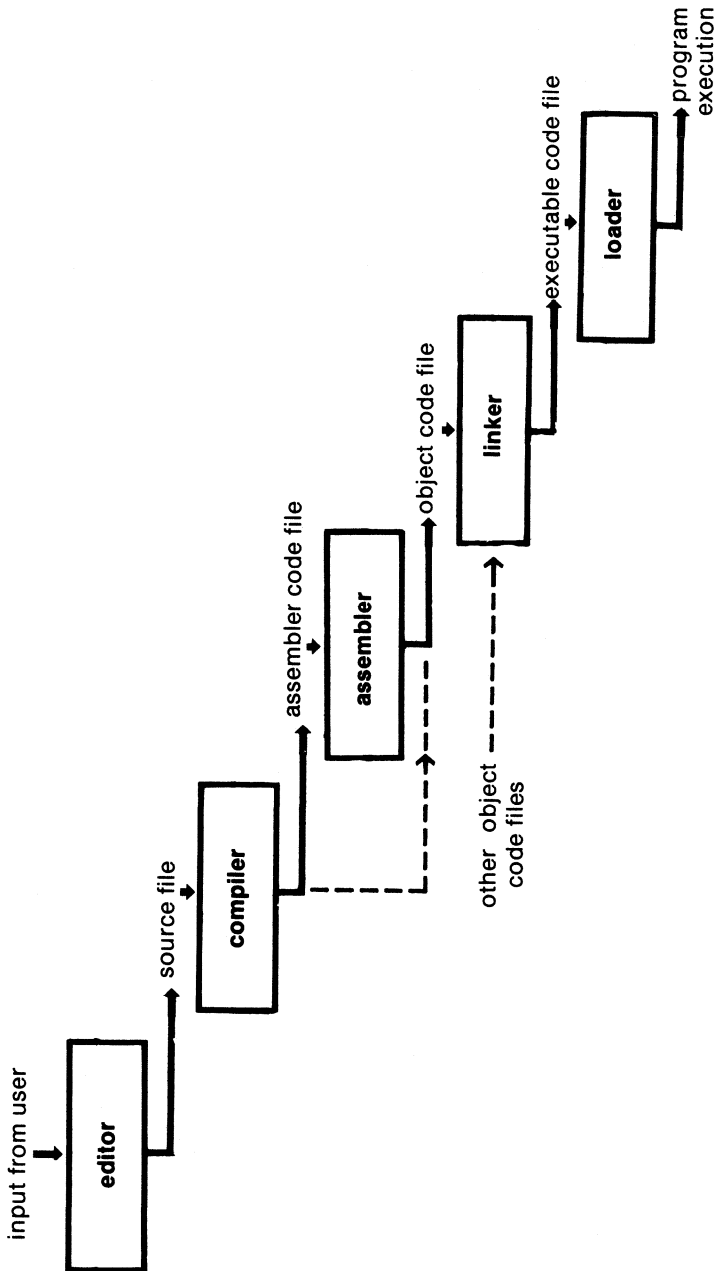
Figure 8.1. Steps necessary to code and run a C program

2. *parse phase* — processes the code output by the first phase. The parser identifies tokens and sections of code and groups these in a scheme to facilitate translation to simpler assembler or machine code.

3. *compiler code generation phase* — translates the grouped code of the second pass into assembly or machine language.

The preprocessor statements of step one are the topic of the majority of rest of this chapter. These statements and the services they perform, though uncomplicated, are very useful. The important point to note here is that this step is completed before proper C language translation occurs.

While it is of course impossible to adequately discuss the operation of compilers in so short a space, there are several points worth mentioning here. First, the final output of a compiler may be either the machine code or the assembly code translation of the source program, depending on the compiler. An assembly language file can be very useful only if the programmer is concerned with working on the most basic level of the machine — perhaps for code efficiency optimization or hardware usage considerations. Since many of the possible low level changes can be directly implemented by C code, often extensive assembly language coding can be avoided.

Secondly, some compilers work in a number of steps, called *passes*, that may or may not correspond to the rough phases outlined previously. Each intermediate pass creates a temporary file that is used by the next compiler pass and is then deleted. (Error detection will normally terminate the current compiler pass, possibly leaving these intermediate files still in secondary storage. File cleanup may then be necessary.) Multipass compilation is implemented quite often on microcomputers because of internal memory space limitations. Compilers typically come with extensive documentation on their operation and libraries.

If the outcome of the compiler is an assembly language file, then this file must be presented to a relatively simple assembler program which translates it into an object code file (sometimes called an *object module*). The object file is mostly in machine language format, however there may be a number of unresolved requests for external variables and functions. This allows the user to compile only part of a program, possibly a function at a time. Each separately compiled section will reside in its own object module. To run such a segmented program, it is necessary to physically combine and properly connect each file section in a process that is termed linking. C library functions are commonly joined to the user-defined program in this fashion.

The *linker* is the program that combines separate object modules and produces the executable machine code file. Only those functions and external variables referenced by main() and its called functions will actually be linked together. For example, a call to the standard function printf() will link just that routine to that program and not the whole standard library, which may be very large. Operation of the linker usually requires the user to list all files that contain externals and functions to be linked, although some linkers search the standard library and current file directory by default. A linker may be provided as part of the compiler package or the operating system linker may be utilized. Linkers do not erase the original object files as these files are commonly reused (e.g. the C standard library).

Finally another operating system program called the *loader* can be invoked to load the program into main memory and start the execution process. This step is implicitly accomplished under many systems by inputting the program's file name with no extension. After program execution, control immediately passes back to the operating system program. Sometimes the functions of a linker and loader are combined into one program termed the *linkloader* or *loading linker*.

The compilation, assembly, linking, and loading steps also have a direct influence not only on acceptable file names, but also on function names. Each of these systems programs is designed to recognize a certain number of specific characters that compose a name. File names must follow system specifications and should not duplicate any system reserved words. Function identifiers must follow both the syntax rules outlined in chapter 2 and any additional limitations imposed by the named system programs. Furthermore care must be taken that user-defined auxiliary function identifiers do not mistakenly duplicate library function names for obvious reasons.

THE C PREPROCESSOR

The C preprocessor provides several unique functions that are unavailable to other high level languages but rather are normally associated with assembly language usage. The preprocessor operates much like a text editor under the control of preprocessor statements or *directives.* Preprocessor directives are coded into the source program before compilation much like "normal" C statements are.

Preprocessor directives also differ from normal statements in form. Directives do not follow the syntax rules for statements. Instead they must observe a smaller set of syntax rules, one of which is that *all preprocessor statements must begin with the pound character, # in column one.* This character is immediately followed by one of the following preprocessor command words, which are not reserved words (# included for emphasis):

#include	#undef
#endif	#ifndef
#else	#ifdef
#define	#if
#line	

Directives do not rely on the semicolon for statement termination.

It is essential that the reader keep in mind that these directives are not language statements in the normal sense of the word. There can be no function calling, no variable manipulation, and only the **#if** directive can make use of expression evaluation. Preprocessor statements may only direct that certain sections of code or strings appear in the extended C source code, while the directives themselves are purged during the preprocessor compile phase. The directives can be divided into three categories: file inclusion, macro substitution, and compiler control.

File Inclusion

It has already been stated that separate object modules can be combined using the linker. The preprocessor directive **#include** performs a similar but simpler service of "linking" together source files. This directive takes the form:

#include *"filename"*

When the preprocessor encounters such a directive, it searches the directory of the original source file for the file *filename*, and then if the file is not found, it searches the "standard" directories. Standard directories are defined by the compiler. Under UNIX and UNIX-like systems, they can be defined by the system manager.

Alternately, the directive:

#include *<filename>*

performs a similar service except that the original source file directory is not searched. Under both, if the named file is located, its contents are copied and inserted at this point into the original file. If the specified file is not found, then an

indicative error will be output and compilation will terminate. #include's can be located at any stage of a program, although this directive is commonly located before all functions.

An example of the use of an #include is shown in figure 8.2. Notice that the filename must be specified in full, including the .C extension that many compilers require. When the file PROG81.C is compiled, the preprocessor searches for OUT-MES.C and replaces the #include directive on line 4a with the entire contents of the latter file, regardless of whether main() requires them or not. Error messages are still keyed from the original PROG81.C file lines; any subsequent error in the OUTMES.C file will be displayed as occurring on the #include line number 4. Compilation of main(), although it results in the compilation of the copy of the contents of the OUTMES.C (in PROG81.C), does not cause the file OUTMES.C to be compiled itself.

If more than one file is to be #include'd in a compilation, each file must have its own #include directive. These may occur either directly in the original compiled source file, or in a nested manner — an #include'd file itself contains one or more #include's. If the file that is an object of this directive is changed, then all the object files that #include this file are obsolete, and the relevant source files must be recompiled.

Macro Substitution

A *macrodefinition* (or more simply a *macro*) is an identifier that symbolizes and is replaced by another, predefined string composed of one or more tokens. The preprocessor directive that accomplishes this function in C is the **#define** statement. There are two different forms of this statement, one for use for simple string replacement and the other for use with argumented string replacement.

In File PROG81.C:

```
            /* Program 8.1 */
 /* exemplifies the use of the #include pre-
      processor statement */
 #include "OUTMES.C"

main()     /* 6a */

{  int er_fl = 0;  /* set to trip */
   /*  . . .  */
   if (er_fl == 0)
       er_mes(er_fl);
   /*  . . .  */
}
```

In File OUTMES.C

```
/* contains functions used to handle output
   messages for selected common situations */

/*  .  .  .  */
er_mes(ind)
int ind;

{  /*  . . .  */
   if (ind == 0)
       printf("\n Division by zero ");
       return (1);
   /*  . . .  */
}
```

Figure 8.2. Example of the use of the #include directive.
Here the included file contains auxiliary standard
functions, such as those used to output error
messages.

SIMPLE STRING REPLACEMENT

Often it is useful to be able to reference constant values
without explicitly writing these values for their every occur-
rence. One method of doing this involves assigning a variable a

value and thereafter referencing said variable. A symbolic constant can also be created through the #define directive of the form:

#define *identifier token-string*

where the *identifier* must follow the rules of names in C. The end of the identifier is taken as the first whitespace character.

After a directive of this sort is encountered, all subsequent occurrences of the specified identifier within that file are replaced with the specified token string, except when that identifier appears enclosed within double or single quotes.

The following line illustrates the use of this directive:

#define PI 3.141592

First notice that the idenitifer **PI** is capitalized. By convention, capitalizing **PI** indicates that the identifier is a symbolic constant. Also note that no terminating semicolon is encountered as in normal statements. If a semicolon had been placed after the identifier, it too would have been replaced along with the indicated numeric value for the identifier **PI**. From the above #define line onward, all occurrences of **PI** outside of strings will be replaced by 3.141592. For example of the following three lines:

```
length = 2 * PI ;
r = hPI + 1 ;
printf(" \n PI") ;
```

only the first would be changed during preprocessing. It would be transformed as:

length = 2 * 3.141592 ;

Constants defined in this manner are sometimes referred to as "manifest constants" because they are given an invariant value

before program compilation and execution.

The token string can be any valid section of C code. The token sting is taken as the string following the second whitespace after the #define keyword. (The identifier follows the first whitespace.) If a backslash is encountered as the last nonwhite character on that line, then the token string continues to the next line. As a matter of good programming style, individual tokens in this string should not be split between lines; each continued directive line should end in a whole token followed by a space then a backslash.

For example, a lengthy decision expression could be symbolized by a macro. Compare figure 8.3 with figure 6.7b. Each program demonstrates a technique for symbolically referencing an expression: figure 6.7b utilizes a function call while figure 8.3 utilizes a *macroprocessor expansion* (i.e. an identifier that denotes or expands to two or more tokens). In the latter case the preprocessor actually inserts the token replacement string for every occurrence of the macro identifier subsequently found in the file. Then the expanded source program is compiled in the normal manner.

Generally a macrodefinition of this sort is considered superior to the analogous function call routine. The advantages include:

- Faster object code execution. The use of macros is equivalent to explicitly coding the replacement string for its every occurrence. Thus the time-consuming operations of argument and returned value passing are eliminated.
- Better conveyence of the symbolic purpose and the related ability to be grouped together.

With regards to object code length, macro use produces more compact code when replacement strings are short and the number of insertions small. Auxiliary function usage produces more compact (but not faster executing) code for the opposite

```
/* lengthly decision expression sym-
   bolized by a #defined macro   */

#define  DEC1   ((a > b !! c > b) && \
    ((d % f) == 0) !! (c == (e = 100 \
    * g) && (g != 0)) && (f != 0) && \
    (h != 1) && ((a + b + c + d)  \
    < 500 ) )

main()
{   int a,b,c,d,e,f,g,h ;
    /* other declarations */
    /* other statements */
    if (DEC1)
        /* if object statement */
    /* other statements */
}
```

Figure 8.3. Illustration of the use of a string replacement macro expansion. This preferred method is in contrast to the use of a function call, as in figure 6.7.

conditions.* Generally the difference in object code length produced between these two techniques is not a major consideration. However if a five line replacement string was to be inserted into a program five hundred times, then the use of the alternative function call technique may well be worth investigating.

Either technique is clearly superior to explicit coding for numerous, lengthy, and/or critical code sections. Both techniques obviously improve source code brevity and clarity. In addition, these techniques facilitate quick and easy program source alterations. Specifically, if the section of the code

---

* Compilation time requirement differences follow the same reasoning, although this aspect is rarely worthy of consideration.

represented by a macro or a function needs to be changed, it need only be done once in these constructions. If on the other hand these code sections were repeatedly coded throughout a program, each occurrence would have to be located and changed — a bothersome and error prone process.

MACROS WITH ARGUMENTS

The preprocessor also allows one to code a more complex but very useful form of the #define. It has the form:

#define *identifier* **(** *identifier , ... , identifier* **)** *token string*

symbolic macro identifier   formal argument list   replacement string (usually includes formal argument identifiers)

Notice that there is no space between the first identifier and the opening parenthesis. (If a space were included at this point, the preprocessor would process this statement as a simple string replacement, with the formal argument as part of the token string.)

This form of the #define directive corresponds to a called function. When this type of macro is implemented, an editorial version of actual to formal argument matching occurs. Of course there are no returned values, but rather an argument edited token string replaces the macro identifier. This form of the #define is for obvious reasons sometimes called a *pseudo-function*.

The best way to explain macros with arguments is through an example. Consider the directive:

#define SQU(x)   x * x

If the following statement appeared later in the same file:

w = SQU(w);   /* s1 */

then the preprocessor would expand this statement to:

w = w * w;

The actual argument **w** was matched to the formal argument **x**. Next, all occurrences of the formal argument **x** in the replacement string were changed to the actual argument **w**. Then this argument edited string replaced the marco call **SQU(x)**.

By the way, interceding whitespace may appear between the macro identifier and the opening parenthesis of the actual argument in the normal source statement. Statement 1 could be coded as:

w = SQU   (w);

This is inadvisable since this macro is designed to mimic a function call.

Multiple actual arguments are matched in an ordinal manner as one would expect. Unlike real function argument matching, the number of actual and formal arguments must be the same. If the corresponding number of arguments do not balance, a compile time error will result.

Because all preprocessor commands perform what are basically editing functions, the programmer must not treat directives as he would normal C statements. This is especially true for the #define directive. Specifically, while expressions used as actual arguments are evaluated before a copy of their value is passed to the called function, no such "evaluation" occurs during argument passing for #define directives. Mistakes commonly arise from improper coding of replacement strings.

Consider the result when the previous macro is applied to the following statement:

w = SQU(w + 1);   /* s2 */

The argument matching, actual substitution within the replacement string, and final insertion of the edited string for the macro call occur as before, only here the actual argument is an expression. The result of this expansion is the following statement:

$$w = w + 1 * w + 1;$$

which is syntactically equivalent to:

$$w = w + (1 * w) + 1;$$

Obviously the resulting expanded statement is not the one intended. The preprocessor blindly performs this code substitution. No compile time error or warning is generated. This shortcoming can be corrected by isolating each occurrence of a formal argument in the token replacement string with parentheses.

$$\text{\#define SQU(x)} \quad (x) * (x)$$

Statement 2 is now correctly expanded as:

$$w = (w + 1) * (w + 1);$$

The perceptive reader may have already surmised that the second version of this macro is still inadequate. To see why, consider the following statement:

$$w = \text{dvdd} / \text{SQU}(w + 1)$$

and the corresponding expanded statement:

$$w = \text{dvdd} / (w + 1) * (w + 1)$$

Since multiplication and division have equal precedence, the outer operations are performed according to the associativity of the operator group, which here is left to right. Statement 3 is

therefore equivalent to:

w = (dvdd / (w + 1)) * (w + 1);

This is obviously not the desired result. Again additional parentheses are the response, but in this case only one pair are required around the entire token string. The correct version of the #define directive therefore becomes:

#define SQU(x)   ((x) * (x))

In summary, each occurrence of a formal argument in the replacement string and the whole replacement string of a #define directive should be explicitly associated using parentheses.

Logical mistakes arising from careless string coding are particularly hard to detect because such errors:

- may only occur for only a few, specific actual argument expression types.
- may occur only when the macro is located in a very specific code environment.
- are more likely to be overlooked in a visual inspection of the program. Experience has demonstrated that programmers verify macro coding only with respect to its logical content and not the environment in which it is used.

MACRO NESTING

Macrodefinitions may be nested; that is a macro may use a previously defined macro in the same source file. For instance, consider the following two macros:

#define PI   3.141592
#define PI_SQU   (PI * PI)

There is no need to surround the value 3.1415 with parentheses since it does not involve operators or variables.

The conditional operator is often used in macros because it is the only expression that allows flow control. Very useful macros can readily be constructed with the use of this operator. Consider the nested pair:

```
#define ABS(x)   (((x)>0) ? (x) : (-(x)))
#define MAX_ABS(x,y)   (ABS(x) > ABS(y) ?
                        ABS(x)  :  ABS(y))
```

where the macro ABS(x) "returns" the absolute value of the argument, and MAX_ABS(x,y) "returns" the largest absolute value of its two arguments.

There are two special points of interest concerning these macros. First, both of them use the parameter x. There is never any dispute in variable uniqueness between macros or between a macro directive and a normal statement. The scope of a macro formal argument only extends to the end of that definition. Secondly note that in ABS(x), the first and second occurrences of x are separated from the operators *within the macrodefinition*. Without such protection, logical errors can occur. For example, if ABS(x) had been defined as:

```
ABS(-ten)
```

where **ten** is a variable with the implied value, then the result of the expansion would be:

```
((-ten > 0) ? (-ten) : (--ten))
```

A problem arises with the compiler's translation of the last term of the expansion. Instead of interpreting this character string as a positive **ten** (i.e. the negative of a negative value), the compiler will translate it as a predecremented variable. Accordingly the above macro call will result in a value of nine instead of ten.

Macro calls can also be nested in much the same fashion as

function calls. For example, if the maximum of the absolute values of the three quantities **100, var1**, and **var2** were desired, the following nested call would suffice:

MAX_ABS(100, MAX_ABS(var1,var2))

THE #undef DIRECTIVE

Sometimes it is useful to end the scope of a #define directive before the end of a file. C provides the **#undef** directive for just such a purpose. This directive has the following syntax:

#undef identifier

where the identifier is the name of a previously #define'd macro. For example, the following directive undefines the macro ABS() for subsequent lines:

#undef ABS

Neither arguments nor parentheses nor any other non-white character may reside on this directive line.

If subsequent to the above #undef, a statement containing a macro call to ABS() is encountered, the preprocessor will not expand such a call. Later compilation steps will treat such an unexpanded macro call as a proper function call. However, if one tries to execute the resulting machine code, an error such as "undefined ABS()" will notify the programmer of the presence of this unexpanded macro call.

In practice this directive is rarely used, since an extant macro is not generally of concern. There are two cases where #undef's may be useful:

- To change the truth value of an **#ifdef** directive. This and related directives will be covered in the section entitled "Compiler Control".
- To resolve disputes in duplicate macro names.

The latter situation, although rare, can occur if a program uses an #include to insert another (usually standard) file's contents into the current file. Files with standard macros are common, so the inadvertent duplication of a macro name in such a case is a very real possibility.

For example, if a hypothetical source program defined the macro ABS(), but also #include'd a file with a macro of the same name, ABS(), then an obvious identifier dispute would occur. Most preprocessors will automatically give priority to the most recently defined macro. In other words, macro redefinition is allowed. However, to avoid any possible compiler dependent warnings or errors, the current file could be coded as:

```
#undef ABS()
#undef ABS(x)    (((x) > 0) ? (x) : (-(x)))
```

Compiler Control

The third aspect of the C preprocessor is known as *conditional compilation*. This attribute enables the programmer, via the preprocessor, to switch on and off sections of code or to decide among a number of alternative program values.

THE #ifdef, #else, #endif, AND #ifndef DIRECTIVES

Sometimes it is convenient to hinge the execution of sections of either normal C code or other directives upon the existence of a key #define directive. C includes the **#ifdef, #else,** and **#endif** directives that fulfill this purpose. The resultant conditional directive takes the following form:

```
#ifdef macro-identifier
  code section
  . . .
#else
  code section
  . . .
#endif
```

where the #else and its object code are optional. The object code is typically comprised of one or more statements and/or directives. The code following the #ifdef but preceding the #else directive is compiled if the indicated macro is currently #define'd. The #else object code is ignored in this case (i.e. not compiled). If the named macro is not currently #define'd, the opposite occurs; the #ifdef object code is ignored while the #else's code is compiled. The #endif directive denotes the end of the construction, as braces have no special meaning to the C preprocessor.

For example, this type of construction is often utilized to insert extra statements to aid debugging of a program. Consider the symbolic section of code presented in figure 8.4. When the **#define DEBUG** directive is present, the section of code between the two directives **#ifdef DEBUG** and **#endif** will be compiled and executed as if all three directives had been absent from the program entirely. Thus a table will be output listing the value of the variables for every loop repetition. Other sections of code in this program could be made conditional to the same macro **DEBUG** in an analogous manner.

If the #define directive had not been present in this program or if an intervening **#undef DEBUG** was present, then all sections of the code between **#ifdef DEBUG** and **#endif** pairs would be disregarded by the compiler. In this example, once all the bugs had been worked out of a program, then the **#define DEBUG** directive could be edited out, and the program recompiled to obtain the basic executable version. The condi-

tionally compiled sections of code should normally be left in the source file since they may again be beneficial during future program revisions.

```
          /* progam start */
#define DEBUG

     .    .    .
{  int sum = 0, prft = 0, run, cntr;

     .    .    .
     for (cntr = 0; cntr < run; ++cntr)
         { process involving declared variables
           .    .    .
#ifdef DEBUG
         if (cntr == 0)
             { printf("\n loop#  \t# shpmts");
               printf("\tsum \tprofit");
             }
         printf("\n    %d \t  %d",cntr,run);
         printf("\t%d \t %d",sum,prft);
#endif
         }

     .    .    .
}
```

Figure 8.4. Example of the use of the #ifdef / #endif directives for conditional compilation of debugging code sections.

The #ifndef ("if not defined") directive is the exact opposite of the #ifdef directive. #ifndef causes subsequent lines to be processed if the indicated macro is currently not defined. The inclusion in C of both directives is largely redundant since the following #ifndef / #else construction:

```
#ifdef identifier
   code section 1
#else
   code section 2
#endif
```

is exactly equivalent to:

```
#ifdef identifier
   code section 2
#else
   code section 1
#endif
```

THE #if AND #else DIRECTIVES

The C preprocessor also includes another more general method of controlling conditional compilation — the **#if** directive. #if-else conditional compilation directives take the following form:

```
#if constant-expression
   code section
#else
   code section
#endif
```

where the **#else** directive and its object code section are optional. The selection of the object code section mimics that for the if-else statement construction except the test expression must be a constant type.

Constant expressions are defined in chapter 4. While variable references are disallowed in constant expressions, references to previously defined macros with constant expression expansions are allowed and, in fact, are quite common. Some preprocessor versions do not allow the use of the conditional operator, the sizeof operator (even when applied to data type operands) or character constants in the constant expression.

Like the #ifdef directive, #if is used to specify compilation of sections of code that are not typically associated with normal user-defined program execution. For example, suppose a programmer wanted to define different security versions of the same basic program. Instead of coding three different versions

```
/* Customer inventory and billing program with
   three hierarchical levels of user priority
   SECUR          max user's ability
   ----------------------------------------
      1           calculate bill, profits, etc.
      2           add delete order entries
      3           output customer orders, addresses
*/

#define SECUR    1, 2, or 3
.   .   .
{ .   .   .
#if SECUR == 1
   printf("\nDo you want to calculate $ amounts?");
   scanf("%c",&ans);
   if (ans == 'y' || ans == 'Y')
       stats();   /* where stats() is a function that
                     calculates accounting quantities */
#endif
     .   .   .
#if (SECUR == 1 || SECUR == 2)
   printf("\nDo you want to add/delete entries?");
   scanf("%c",&ans);
   if (ans == 'y' || ans == 'Y')
       edit();    /* where edit() allows deletion &
                     addition of customer orders */
#endif
     .   .   .
}
```

Figure 8.5.   Example of the use of the #if directive in conjunction with a #define.

of the program, #define and #if directives can be used to alter the executable version of said program.

A symbolic representation of this scheme is represented in figure 8.5. Depending on the value "assigned" to the macro **SECUR**, the executable version of this program will contain correspondingly appropriate sections of code. For example, if **SECUR** was assigned a value of 2, the three statements following #if **SECUR** = = 1 would not be compiled into the executable program.

The appropriate functions may likewise be conditionally compiled. Of course for such a scheme to succeed, a method must be employed to ensure that neither the source code nor privileged executable versions of the program fall into un-authorized hands.

NESTED CONDITIONAL COMPILATION

The conditional compilation directives can be nested in a manner similar to if-else statements. During nesting, an #endif directive must be present for each #if, #ifdef, and #ifndef directive. These will appear in the same relative position as the closing brace of the object block in the analogous if-else construction.

```
#ifndef  MAX
    object line(s)
#else
#if  MAX == 100
    object line(s)      /* 5 */
#else
#if  MAX == 1000
    object line(s)
#else   /* default case */
    object line(s)      /* 10 */
#endif
#endif
#endif
```

Consider the above nested directive construction. It is a third level nested conditional compilation directive. There are a number of diverse points exhibited by this section of code that are worthy of mention:

- The #if, #ifdef, and #ifndef directives can be mixed within a construction.

- The three #endif's at the end of this construction signify this is a third level nested directive construction. Therefore there must be at least three "if" type directives in the construction.
- Care must be taken with the placement of comments. Inclusion of a comment on the same line as a #define will cause inclusion of that comment as part of the corresponding macro expression in most preprocessors.
- Nested simple "if" constructions are not allowed on many preprocessors.
- The # character must be located at column one. For this reason the #else and the subsequent #if (or other) directive keyword must occupy separate lines.

Athough the # character must occupy column one, some compilers allow the remainder of the directive keyword to be indented an arbitrary number of blank spaces. This is often used to offset object directives as in the following lines:

```
#ifdef MAX
#    define MAX 100
#else
    . . .
```

The reader should keep in mind that this is a non-portable attribute.

PREPROCESSOR COMMAND FLAGS

The Unix operating system and some compilers provide compilation options. These options can be engaged by command line "switches" or "flags'. There is one option that is particulary useful with regards to preprocessor directives —the "-d"switch.

To change the truth value of a #ifdef or #ifndef directive, it is usually necessary to edit the relevant source file. However,

in systems that support the -d switch, the programmer is able to #define a macro on the command line for compilation. A macro #define'd using the -d switch is effectively inserted at the beginning of a program during the preprocessor phase of compilation. The actual source text is left untouched.

For example, suppose that hypothetical file sample.c contained several **#ifdef DEBUG** constructions, but no corresponding **#define DEBUG** directive. The #ifdef's could be tripped by either inserting the previously mentioned leading #define directive or by the use of the -d switch. Implementation differs between systems. but the relevant command under Unix would be:

cc sample.c -D DEBUG

while on the Computer Innovations C86 Compiler® it would appear as:

cc1 -dDEBUG SAMPLE

With Unix it is also possible to assign simple macro replacement strings on the command line. For example, the following command line defines the macro **DEBUG**, assigns it a value of one, then compiles the source file:

cc sample.c -D DEBUG=1

Almost all compilers support some types of extra options that the user should be aware of. For example, the C86 also has a "**-p**" switch that outputs the expanded source code. The user should check his or her compiler documentation for these system dependent options.

Line Control

Actually there is one other minor function provided by the preprocessor — namely that of line reference control. In

explanation, it is possible to specifically cause the compiler to institute an artificial numbering scheme and source file name. These changes are only reflected on compiler error and warning messages. The **#line** directive causes such a change. It has the following syntax:

#line *int-constant identifier*opt

When a line of this sort is encountered, the preprocessor associates the next source line with the indicated integer constant. Line numbering continues from that point on utilizing the indicated baseline constant. If an identifier is also present, the reference file name also changes with regards to compiler messages.

To illustrate this directive, consider figure 8.6. Program 8.2 contains ten lines of normal source code. The compiler would normally reference these lines as 1 thru 10. But because of the **#line** directive on line 7, the compiler renumbers the subsequent lines in accordance with the directive. Instead of being referenced as line 8 of PROG82.C, this line is referenced as line 300 of QUACK.C. The next line is then 301 of QUACK.C, etc. This renumbering and renaming scheme in no way affects program execution. The **#line** directive is very rarely used.
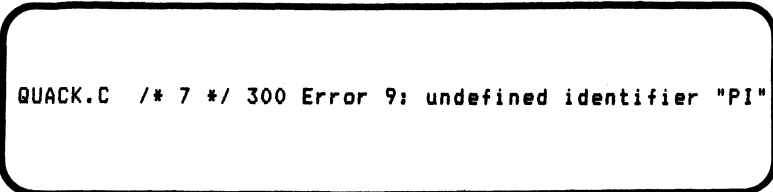
```
/* Program 8.2, File PROG82.C */
/* demonstrates line reference control pro-
   vided thru the #line directive */

main()               /* 5 of PROG82.C */
{  int a = 150;
#line 300 QUACK.C    /* 7 */
   a = 150 * PI;     /* 300 of QUACK.C */
   printf("\n\t %d ",a);
}                    /* 302 */
```

program output on following page

```
QUACK.C  /* 7 */ 300 Error 9: undefined identifier "PI"
```

Figure 8.6. Example of the use of the #line directive to change the line reference scheme of the compiler.

Transportability and Flexibility Considerations

One very important use for preprocessor directives is in the development of a scheme to further the design aspects of a program, especially transportability and flexibility. System or application dependent values or operations can be implemented thru a macro. When the circumstances change, it is a simple matter to alter the relevant macros to conform to the new requirements. This adjustment process is aided greatly by proper documentation, including comments.

Consider the following instances. It is often very useful to know the largest and smallest possible value corresponding to each data type. These values are dependent on the number of bytes (and also the representational format) allocated to each variable of the data type. On the reference machine used with this book, int's can range in value from +32767 to -32768. Accordingly two useful macros can be defined and used on the source system.

```
#define HIGH_INT_VAL 32767
#define LOW_INT_VAL (-32768)
```

If a program containing these lines is subsequently recompiled on a different machine, the values associated with the macros have to be changed. For example, the IBM 360® family of computers uses four bytes to store int values. There these

macros should appear as follows for programs run on those mainframe machines:

```
#define HIGH_INT_VAL 2147483647
#define LOW_INT_VAL (-2147483648)
```

Macros can be similarly used to define program specific values. It is very common to vary the maximum number of allowed input values. This logically leads to the implementation of a macro such as:

```
#define MAX_DATA_VALS integer
```

Macros such as these are often used to specify the number of elements in an array (as discussed in the next chapter), as in:

```
int in_data[MAX_DATA_VALS];
```

This statement would declare **in_data** to be an *integer* x 1 dimensional array.

These two sets of examples have used simple string replacement macros. It is of course possible to extend this reasoning to macros with arguments. Consider the problem of obtaining the lower order half (of bits) of an integer. One appropriate directive set would be:

```
#define INT_BITS 16
    .   .   .
#if INT_BIT == 16
#define R_HLF_BITS(x) (((x) << 8 >> 8) & 255
#endif
```

where the conditional compilation directive is included to ensure that R_HLF_INT() is changed when and if the value of INT_BITS is. (This is a small example of *defensive programming* — the addition of often unnecessary or redundant failsafe code.) The macro INT_BITS would have been used directly in R_HLF_BITS to make the latter portable, however the value

255 is nonportable. The value 255 is binary 11111111 and is AND'd with the result of the shifts to guard against possible arithmetic right shift.

HEADER FILES

When macros are used repeatedly, it is often convenient to group related ones together in a macro library file. This type of file is termed a *header file* and by convention carries a file extension of **.H.**. For example, most C library versions contain the file STD1O.H. This file contains all the *st*andar*d* macros used for *i*nput and *o*utput operations. The standard C library "functions" getchar() and putchar() (chapter 11) are commonly implemented as macros.

After the desired macro source file has been created, it can be easily #include'd in any program. The header file must, of course, be in the proper directory for the #include to be executed successfully. For example, suppose one created a file with a number of #define'd scientific constants, unit conversions, and simple pseudofunction definitions. If this file was named SCIENCE.H, its directives could be used in any program by beginning the program with one of the following lines:

#include "SCIENCE.H"

#include <SCIENCE.H>

File Organization

At this point the reader has been introduced to the different types of C code lines and sections of code. The question now arises as to how a file with a C program should be organized. A general convention has arisen that is quite logical and workable. The corresponding model of this convention is presented in figure 8.7. The reader should keep in mind that the convention is meant as an approximate guideline and that minor deviations may be necessary.

```
/* file and/or program comments */

/* directive section comments */
#directives
      -#includes
      -#undefines
      -#defines
      -conditionals (#if, #ifdef, #ifndef,
                        #else, #endif )

/* external variable comments  */
external variable declaration and initialization
  .     .     .

/* function comments */
function-identifier(argument, ... ,argument)
argument declaration
  .     .     .
{  C program statements
      -external function redeclarations
      -external variable redeclarations
      -local variable declar and initln
      -program logic statements

   -optionally interspersed with :
      -/* specific peicewise comments */
      -conditional compilation directives
      -indented subblocks (optionally with
                        the above body elements)
}

other functions .   .   .
```

Figure 8.7. Symbolic model of the preferred method of inter-file organization.

A Closing Note

The use of directives is very straightforward as long as the programmer remembers that they can themselves, only perform editing operations. The code, if any, that results from these editing operations is what affects execution of a program. Typically preprocessor generated error messages are more clear and easier to correct than their normal C code counterparts. However the programmer should be careful to avoid circular #define's and #include's, such as the following:

#define NUMBER NUMBER

Also indirectly circular #include's arise when an included file in turn includes the original file. Circular directives often result in catastrophic errors and/or confusing error messages.

9

Pointers and Arrays

Introduction

This chapter introduces two new and very important, non-fundamental data types — the pointer and the array. The word "pointer" connotes, in computer science, a data value that indicates a location in main memory. Pointer variables, therefore, point to one or more bytes in memory. Pointers are used very extensively in assembly language programming. Pointers also play a very important role in C.

Arrays are a data type found in most higher level languages. The array allows the storage of a number of data values in one variable. The underlying concepts of arrays are derived from matrix algebra. Besides being used to represent matrices, arrays are also suited for a variety of other purposes because, in C, this data type represents a defined number of *contiguous* (i.e. consecutive) bytes in memory. There is a very strong relationship between pointers and arrays in C that will become more evident as this chapter progresses.

The Concept of Derived Data Types

Pointers and arrays are both examples of *derived data types* in C. A data type is said to be derived when it is "built upon" one or more of the fundamental data types optionally in conjunction with operator(s) to form a new *object*. In computer science the word object refers to any identifiable, complete division in memory that can be manipulated. C allows the following derived object types.

- *functions* returning a fundamental data type, or pointer. Unix Version 7® added the ability to return structure.
- *pointers* to a fundamental data type, pointer, array, structure, union, or function.
- *arrays* of fundamental data type objects, pointers, structures, unions or other arrays.
- *structures* of fundamental data type objects, and/or pointers, and/or structures, and/or unions. (chapter 10)
- *unions* which may contain similar objects to structures. (chapter 10)

All of these objects, except functions, are derived data types. Of these, all except the pointer data type are *aggregate* objects. An aggregate data type is comprised of two or more constituents, each of which may be a fundamental data type

object or another permissible derived data type.

Arrays, structures, and unions may even be defined in a nested manner. That is, one can have an array of arrays, a structure containing a structure, or a union containing a another union.

Mechanism of Variable Use

Pointers are perhaps the most confusing aspect of C to programmers who have had no past exposure to a low level language. This difficulty often exists because most of these programmers are not aware of how variables actually function. The variable-as-a-box model is adequate for the novice, but it quickly becomes inadequate, because it fails to account for the way in which the compiler and the operating system handle variables.

By the time a program is translated into machine language, all variable references have been purged. The compiler (or interpreter) associates the proper variable name with a selected, unique address in memory.* The compiler also allocates the number of bytes after that address as suggested by the variable's data type declaration. In the machine language code, this address substitutes for the proper variable name. Whenever a programmer specifies a variable in a statement, what he or she is actually doing is referencing a memory address.

From this discussion, it is obvious that a variable (as well as any other object) has at least two attributes associated with it: an address and the data associated with that address. This associated data occupies the bytes immediately following the indicated address. For example suppose during execution of a

---

* Actually only an index address (and displacement for aggregate data types) is chosen by the compiler. These are added to a base address-selected by the operating system at run-time — to give the actual memory address. The ability to select a base address means the operating system can locate the program wherever it wishes in memory; the executable program code is said to be *relocatable*.

program, the following declaration (or its machine code equivalent) was encountered:

int sum = 100;

The operating system would allocate, somewhere in memory, the appropriate string of bytes for this variable. For the purpose of this discussion, assume an int is stored in two bytes and that the operating system has chosen address location 10,000 and 10,001 for **sum**. Then during this execution of the program, every mention of the variable **sum** is translated in the machine code to the address 10,000. Since the data type of the variable is declared as integer, normally all generated instructions will manipulate the indicated two bytes as pure binary data.

The concepts of a variable's memory address and the contents at this address correspond to *lvalue* and *rvalue*, respectively. An lvalue is the memory address where data associated with a conceptual quantity is stored. In higher level languages, actual numeric lvalue locations are normally replaced by symbolic lvalues which are variable identifiers. The rvalue is the data that resides after this location. When **sum** is initialized to one hundred, the two byte binary representation of this value is stored at locations 10,000 and 10,001.

Pointer variables have a quite ordinary lvalue associated with them. What is unique about a pointer is that its rvalue, or the held value, does not represent a physical or conceptual quantity, but instead represents a memory location. Thus a pointer points to another location in storage. When a pointer's rvalue is set to the beginning address of another object, the object can be referenced indirectly by a process which we will explain next.

Declaration and Initialization of Pointers

The declaration of a simple pointer variable takes the following form:

data-type *identifier ;

This declaration defines three attributes of a pointer variable:

- The relative (relocatable) address or "actual" lvalue of the pointer variable that is associated with the symbolic lvalue *identifier.*
- The rvalue associated with this address is an integral pointer value.
- The number of bytes associated with the data type being pointed to by the pointer's rvalue is defined by *data-type.*

Notice that the first two attributes are similar to fundamental data type declaractions but the third is a new attribute for pointers.

To clarify this process, consider the following declaration:

int * sum_ptr;

This statement declares a pointer variable named **sum_ptr**, which points to an integer data type. There are two important points to stress. The * or the indirection operator does not perform its normal function in a declaration. Instead it merely defines **sum_ptr** as a pointer variable. Secondly the specified data type, here int, is the data type of the object being pointed to by **sum_ptr** and *is not the actual pointer's data type. The correct way of interpreting this statement is "declaration of* **sum_ptr** as a pointer to an int value". Pointer variables themselves are stored in a system dependent manner, generally as int, unsigned int, or long int. The range of the data type that pointers are stored as must be sufficient to accommodate the largest memory address of the computer.

Pointers are commonly assigned values through the use of the "address of" operator, **&**, applied to a variable. For example, the following statement assigns to **sum_ptr** the address of the variable **sum**, which was assumed to be 10,000.

<div align="center">**sum_ptr = ∑**</div>

Assuming that the pointer variable is stored as an init and that it has been allocated two bytes beginning at memory location 11,500, the situation can be represented as follows:

The contents of **sum_ptr** points to the starting address of **sum**. (The rvalue of **sum_ptr** is the lvalue of **sum**). The compiler knows that two bytes (on this system) are associated with this pointing because **sum_ptr** was declared with the data type int. The data type also specifies the data representation scheme used to store the data.

Pointer variables can be assigned a literal value. Systems programmers may do this because the operating system is always loaded in the same main memory block. Applications programmers cannot use this convenience unless they also reference the operating system (which may be protected).

The Indirection Operation

Before proceeding, a matter of nomenclature must be resolved. Technically an elementary expression such as **&**_identifier_ is known as a _pointer value_, while a variable such as *_identifier_ is known as a _variable pointer_. In practice, however,

both are commonly referred to as "pointers".

Once a pointer value has been stored in a pointer variable, the question remains as to how to reference the object being pointed to via the pointer variable. The answer is by way of the unary *indirection* operator, *. When applied to a pointer variable in a statement other than a declaration, the following steps occur:

1. The pointer variable is located in memory (i.e. its lvalue is established).

2. The pointer variable's rvalue is referenced. This value is taken as an address of the data type named in the pointer's declaration.

3. The object pointed to by the variable pointer is located. This object is the result of the indirection operation.

For example, consider the following statement:

```
double tax = 0.0;
int sum = 100, *sum_ptr = &sum;
tax = *sum_ptr * 6.5;
```

Note that the declaration and initialization of the pointer **sum_ptr** has been accomplished on the same line as the declaration of its "target" variable sum. This is valid as long as the target variable is declared first. The third line contains the indirection expression ***sum_ptr**. This expression is evaluated in the following manner:

1. **sum_ptr** is located in memory.

2. The rvalue of **sum_ptr** (assumed to be 10,000 previously) is obtained.

3. The int value at this address location, which is the rvalue of **sum** or 100, is referenced. 100 is the value "returned" by this indirection operation.

The statement proceeds by type converting the integer 100 to

its double equivalent, multiplying this value by 6.5, then assigning the result to **tax**. Observe that the actual addresses of these variables do not explicitly appear, and are inconsequential in the programmer's view.

Before continuing, it is important that the reader have a firm understanding of pointers and the indirection operation. Indirection is the process of accessing the contents (rvalue) of a target variable through the associated pointer. The indirection operator, *****, can be translated as "that which is contained at the memory location specified by ...".

Pointer Arguments and Privacy Exception

While reading the last section, it might have occurred to many readers that the **tax** assignment line could have been accomplished in a more direct and efficient manner, without the use of a pointer process, as in the following statement:

tax = sum * 6.5;

This is indeed the case. However the major reason for using pointers to (local) variables is that through their use, the privacy of local variables can be overridden. In other words, if pointer values to local variables are passed to a called function, that called function can alter the actual values of the calling function's local variables through indirection.

One of the most basic examples of the use of pointers to override function privacy is the C library function swap(), presented in figure 9.1.

A call to swap would appear as:

swap(&*variable-a,***&***variable-b***)**

The addresses of the two variables are passed to swap() and matched to pointer variables **p_a** and **p_b**. The value of the first is assigned to the local (temporary) variable **tmpry** in line 6.

```
/* interchange 2 int values */
swap(p_a, p_b)
int *p_a, *p_b;

{   int tmpry;
    tmpry = *p_a;
    *p_a = *p_b;
    *p_b = tmpry;
}
```

Figure 9.1. The library function swap() is a prime example
of the use of pointers to violate function privacy.

Then line 7 assigns the contents of the second variable (i.e.
variable-b) to the first variable by a double indirection opera-
tion. Line 7 can be interpreted as "assign the value located at
the address pointed to by **p_b**, to the location pointed to by
p_a." The indirection operator is one of the few non-primary
operators that can legally appear on the left side of an assign-
ment operation. Line 8 finishes by assigning the value of the
first variable, which was stored in **tmpry**, to the address loca-
tion of the second variable.

This same routine could not have been accomplished
through one function call without the use of pointers. Nor-
mally a called fuction can only affect the values of the calling
function's local variables indirectly through one returned
value. Swapping demands two such values. Aside from pointer
usage, only a macrodefinition would have allowed this process
to succeed through one "function" call.

Think back to the scanf() function. A call to this function
also passes pointer values of this calling function's local varia-
bles. This input function then assigns the proper values to the
respective calling function's variables through indirection. The
use of pointer values, variables, and operations (i.e. address of

and indirection) in this manner greatly reduces the need for external variable usage, which in turn generally increases modularity and portability.

Pointer Conversions

Pointers may to a limited degree be type converted, however the process is largely machine dependent. As was true for the fundamental data types, data type conversion can occur across an assignment operation, or it can be explicitly coerced through the use of the cast operator.

A pointer to a wider type can normally be converted to a pointer to a thinner type and back again without causing problems. In constrast, the inverse may result in difficulties because wider data types have more stringent alignment requirements. As noted in chapter 3, a variable is aligned in main memory at an address that is a multiple of the number of bytes associated with it. On this text's reference system, char's are aligned on any bytes, while float's are aligned on adresses that are multiples of four. Consequently conversion of a pointer to a char to a pointer to a float will only succeed if the character that was originally pointed to was aligned at an acceptable float address (i.e. an address of the form 4n, where n is a positive, in range integer). If the character was not aligned in such a manner, a special error termed an *address exception* will occur.

A pointer value may also be converted to an integral data type large enough to contain it. This is machine dependent as pointers may have the same memory allocation as either int (or unsigned int) or long int. The way in which the main memory addresses are arranged and represented numerically (sometimes referred to as the *memory mapping function*) is also machine dependent.

An integral value can be converted to a pointer successfully, assuming that the limits of a pointer of the particular

machine are great enough to contain such a value. A conversion of this type can occur across an assignment or by the cast of the type:

$$(\textit{data-type} \; *) \; \textit{integral-identifier}$$

In general it is rarely necessary to convert a pointer to another data type because a certain subset of arithmetic operations are allowed on pointer variables. An examination of these operations will be delayed until after arrays have been discussed since pointer arithmetic is usually associated with array (or structure) usage.

Pointers and Functions

Only two operations can be performed on a function in C:

- A function may be called from another function or from within itself.
- The address of a function may be assigned to a pointer variable. The pointer variable may then be handled as any pointer.

The second operation makes it possible to *effectively* store functions in arrays and structures, to pass functions to other functions, etc.

If the function, funct( ), has been previously defined in the source file or if it has been redeclared in the current function, then a token of the form:

funct

yields a constant pointer value to the beginning of that function. Such a value can be assigned to a properly defined pointer variable, which has the form:

$$\textit{data-type} \; (*\textit{identifier})();$$

This statement is interpreted as "declaraction of the *identifier* as a pointer to a function returning *data-type*". For example,

double (*ptr_funct)( );

declares **ptr_funct** to be a pointer to a function returning a double precision floating point value. (The first pair of parentheses are required to associate the indirection operator more closely to the identifier than the argument parentheses.) The assignment of the function pointer value to the pointer variable can be accomplished through the statement:

ptr_funct = funct;

while the indirect call to this function can be accomplished through the statement:

(*ptr_funct)(*argument list***);**

Functions that return a pointer value should also be specially declared. The statement:

data-type ******identifier***( );**

declares the function *identifier*() as a function returning a pointer to *data-type*. For example,

long int * m_val( );

defines m_val( ) as a function returning a pointer to a long int.

One word of warning: In some programs pointer values are often matched to integer formal arguments, and returned pointer values are returned as integer values. This strategy of treating pointers as int's will succeed on many machines because pointers are generally stored in the same format as int's. However this is both a non-portable and careless practice and should be entirely avoided.

Notion of Arrays

C allows the definition of three main types of derived aggregate data types for handling groups of related data. When the data set is composed of an orderly arrangement of values of the same data type, this set can be conveniently represented in an array form.

Each constituent value of an array is called an *element*. When an array is defined in C, the programmer specifies the number of elements in the array. The elements of an array can be ordered in different pattern schemes. A *one-dimensional* array can be conceptually viewed as a single string or row of a predefined number of values. For example, a linear array of seven integers could be envisioned as:

column	0	1	2	3	4	5	6
row 0	int1	int2	int3	int4	int5	int6	int7

In array notation this arrangement is known as a 1 x 7 array because it consists of a single row and seven columns. In C the elements of a dimension are numbered from zero to n-1, where n is the number of values associated with that dimension.

An array including more than one dimension is called a *multi-dimensional* array. For example, a 3 x 7 integer array could be conceptually depicted as shown in figure 9.2.

column	0	1	2	3	4	5	6
row 0	int1	int2	int3	int4	int5	int6	int7
row 1	int8	int9	int10	int11	int12	int13	int14
row 2	int15	int16	int17	int18	int19	int20	int21

Figure 9.2. An example of a 3 x 7 array.

Each of these elements is associated with a column number in the range of zero to six and a row number in the range of zero to two. These positional values are known as *subscripts* or *indices*. If the above array is named **example** then *int12* can be identified as **example**$_{1,4}$. Notice that the total number of elements is the product of the number of elements in each dimension, here 21.

In conclusion an array in C can be differentiated by four attributes:

- Number of dimensions or dimensionality.
- Number of elements in each dimension.
- Data type of the elements.
- Actual value of each element.

The first two attributes determine the "size" of the array in the conventional mathematical sense, while the first three attributes determine the amount of required memory storage.

Array Declaration and Element Referencing

Declaration of linear arrays in C takes the following form:

data-type identifier **[***subscript***]**;

This defines *identifier* as a *subscript* x 1 array where each element is of the type *data-type*. For example, the following line declares **fl_ary** to be a 200 x 1 array of float's:

float fl_ary[200];

The dimension size must be a constant integer expression. C does not inherently provide a convenience called *dynamic allocation*. Dynamic allocation allows the programmer to use a variable as the dimension size during the declaration of an array. Then the array size is determined during program execution. (The system dependent C library functions calloc() and

free( ) can be used to achieve this effect in C, however only with considerably more effort on the part of the programmer.)

In other statements, an element of an array can be referenced separately by enclosing the desired element subscript in brackets following an array identifier. (By convention no whitespace is inserted between the identifier and opening bracket.)

identifier[element#]

While the use of variable expressions as the dimension size in array declarations is forbidden, variables are allowed and commonly used to reference individual array elements (i.e. variables as indices). C does not keep a close check on element referencing. It is perfectly legal to reference an "element" beyond the proper bounds of an array, although the referenced value is normally unknowable (i.e. garbage). For example, given the previous array declaration, the following two expression are completely valid:

fl_ary[375]
fl_ary[-17]

The reader should keep in mind that the proper limits on an array are from 0 to n-1 where n is the declared size of an array. For example, the proper elements of **fl_ary** would be referenced as:

fl_ary[0], fl_ary[1],...,fl_ary[199]

Multi-dimensional arrays are declared in a similar manner, and have the form:

data-type identifier[subscript][subscript]...;

For instance, the declaration,

int two_dim[3][2];

defines **two_dim** as a two dimensional array, specifically a 3 x 2 array. The total size of an array is the product of all the array elements. Therefore the overall size of **two_dim[ ]** is 3 * 2 ⇨ 6 int elements, which on the reference system occupies 12 contiguous bytes.

    Multi-dimensional array objects are arranged so that the rightmost sub-array or dimension forms the most basic repeating unit. Equivalently, as the memory address increases, the subscript to the rightmost dimension changes most rapidly. If coordinates that correspond to the appropriate subscript are assigned to each array element, the 2-dimensional array **two_dim[ ]** would appear in memory as depicted in figure 9.3a. Notice that the rightmost dimension, here the column subscript, changes most rapidly, while the first or row subscript is changed the least often.

    Array organization of this type is necessary because memory appears as a linear arrangement to the user. Hence a n-dimensional array must be stored in a linear fashion as just demonstrated. Compare the (geometric) conceptual presentation of **two_dim[ ]** which appears in figure 9.3b with the arrangement of this array's elements as shown in figure 9.3a. The memory implementation of arrays in C will be of particular relevance when the relationship between pointers and arrays is discussed.

    This logic can easily be extended to higher dimensional arrays. Again the rightmost subscript varies most rapidly, the next to the rightmost varies second most rapidly, etc. However, since this is a three dimensional world, the conceptual interpretation of arrays breaks down when the dimensionality exceeds three.

    One method of referencing array elements relies, as just demonstrated, on the geometric conceptualization of arrays. Hence, if an element is described as occupying the fifth row and one hundred and first column of some hypothetical array, then the C subscript notation for this element would be **[4][100]**. Under this scheme, array element references range from

a)
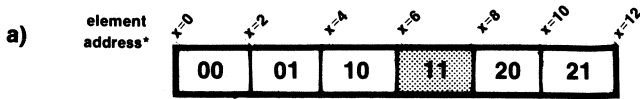
element
address*

*where x is the starting address chosen by the operating system prior to program execution. The int data type is assumed to have a memory allocation of two bytes.

b)
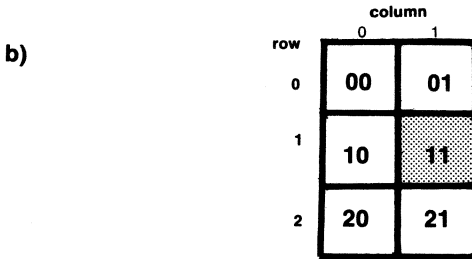
Figure 9.3. a) Internal storage implementation of array **int two_dim[3][2];**. The value held by each element corresponds to its subscripts. For example, the shaded box's value can be referenced by the expression **two_dim[1][1]**.

Figure 9.3. b) Conceptual representation of the same array.

identifier[0][0]... to *identifier* [*m-1*][*n-1*]... *where identifier* is the array name, *m* is the declared row size, *n* the declared column size, etc.

The other method of referencing the elements of an array relies on the fact that C does not monitor the reference subscript for out-of-bounds conditions. Therefore it is possible in C to reference any array as if it were a linearly arranged set of objects. This method is directly related to the actual arrangement of elements in memory. For example, from figure 9.3a it is obvious the element **two_dim[1][1]** is the fourth element in memory. This element could equivalently be specified by the expression **two_dim[0][3]**. Under this method, proper element references range from *identifier*[0][0]...[0] to *identifier*[0][0]... [*product* - 1] where *product* is the product of all the declared

dimension sizes. As an instance, our model array's elements range from **two_dim[0][0]** to **two_dim[0][5]**.

The reader should keep in mind that the initial subscript in both schemes is represented by a value of zero and not one.

Operation on Arrays

Only two operations may be performed on an entire array; the address of an array may be taken and an element of an array may be referenced. (Some compilers also allow the sizeof operator to be applied successfully to an entire array.) An array name without the subsequent bracket pair represents the beginning address of the specified array. An array element can be referenced by the token string composed of the array name followed by a bracket pair enclosing the appropriate subscript(s).

Both of these operations are exemplified in Program 9.1, presented in figure 9.4. Line 6 declares **ary** to be a 5 x 1 array of integers, **p_int** to be a pointer to an integer, and **p_ary_int** to be a pointer to an array of integers. The last two variables are assigned the address of the array **ary** in lines 14 and 15 respectively. Technically the declaration of **p_ary_int** (i.e. pointer to an array of int's) is the correct one. However since arrays represent a linear collection of their elements, a pointer to the simpler element data type (here pointer to int) is sufficient, and indeed more commonly used.

Lines 8 through 12 are simple assignment statements to the elements of **ary**. Note that the assignment of element values may occur in any order. Individual array elements of a given data type and storage class may be used wherever their non-array counterparts can. In Program 9.1, the array elements are of auto int flavor and can be utilized like any such flavored variable.

```
              /* Program 9.1 */
/* illustrates the allowable entire
        array operations         */
main()
                            /* 5 */
{   int ary[5], *p_int, (*p_ary_int)[];
    int index;
    ary[1] = 2;
    ary[4] = 7;
    ary[2] = 3;              /* 10 */
    ary[0] = 1;
    ary[3] = 5;

    p_int = ary;
    p_ary_int = ary;     /* 15 */

    for (index = 0; index <= 4; ++index)
        printf("\n ary[%d] = %d ", index,
                            ary[index]);
    printf("\n\nary = %d, \t &ary[0] = %d",
                            ary, &ary[0]);
    printf("\n\tp_int = %d", p_int);
    printf("\n\tp_ary_int = %d", p_ary_int);
}                            /* 24 */
```

```
  ary[0] = 1
  ary[1] = 2
  ary[2] = 3
  ary[3] = 5
  ary[4] = 7

ary = -82,          &ary[0] = -82
          p_int = -82
          p_ary_int = -82
```

Figure 9.4. Program 9.1 illustrates the addressing of an entire array and element referencing.

Observe that the for loop on lines 17 through 19 utilizes the variable subscript **index**. This in no way conflicts with the requirements that arrays must be declared with a constant size subscript. The output verifies not only the aforementioned points, but also the relationship of the notations **ary** and **&ary[0]**, which is further explained in the section entitled "Array and Pointer Notation Equivalence". The actual value of the beginning memory location of **ary** during this reference run is highly system and circumstance dependent.

Array Initialization

Some general rules for the initialization of arrays follow:

- register arrays are disallowed.
- Only static and extern arrays may be explicitly intialized.
- Automatic arrays are always implicitly initialized to whatever previous values were in the memory blocks allocated to the elements (i.e. auto arrays are initialized to garbage).
- Any static and extern array element not explicitly initialized is implicitly initialized to zero.
- There are no value repetition factors in C as there are in many of the high level languages.
- Too many initializing values (i.e. more values than array elements) will produce a compile time error.
- Each array *initializer* (i.e. the token(s) used to initialize an element) must be an expression resolvable to a value of the same data type as the array elements.

static and extern linear array initializers have the following form:

$$= \{ \; constant\text{-}expression,...\};$$

where the comma separated constant values are termed the *initializer list*. The first constant expression is evaluated and

assigned to the first array element, which is *identifier*[0]. The rest are likewise ordinally matched. If there are fewer initializers than array elements, then the unmatched elements are initialized to zero. For example,

```
static int prime[7] = { 1,2,3,5,7,9 };
```

declares **prime[]** as a 1-dimensional array of seven integers. The first six values are explicitly initialized as 1, 2, ... 9 respectively, while the seventh element (i.e. **prime[6]**) is initialized to zero by default.

The preferred initializer for multi-dimensional arrays has the form:

= {{*initializer-list*}, {*initializer-list*},...}

where the initializer list is usually a comma separated, optionally further brace-grouped series of literals or constant expressions.

Braces are used to denote a dimensional level. For example the separate rows of initializers would be grouped as follows:

```
static int two_dim[3][2] =
        {{0,1}, {10,11}, {20,21}};
```

In this manner element **two_dim[0][0]** is intialized to 0, **[0][1]** to 1, **[1][0]** to 10, **[1][1]** to 11, **[2][0]** to 20, and **[2][1]** to 21.

If a value is not explicitly specified, the matching element is initialized to zero. For example, the declaration:

```
static int keep[2][3][5] =
    {{{1}, {2,3}, {4,5,6}},
     {{7,8,9,10}, {11,12,13,14,15}, {0}}};
```

assigns the values 1,0,0,0,0 to elements **[0][1][0]** through **[0][0][4]** respectively; 2,3,0,0,0 to elements **[0][1][0]** through **[0][1][4]**; 4,5,6,0,0 to **[0][2][0]** through **[0][2][4]**; 7,8,9,10,0 to **[1][0][0]** through **[1][0][4]**; etc. Notice that the inner brace pairs group the values as expected — namely into two groups of three subgroups each, with each subgroup conceptually containing five constant values. Where five values are not explicitly stated, implicit zeroes are added. (For example, {2,3} is translated by the compiler as {2,3,0,0,0}.)

If more values appear in a list than necessary for the current dimensional level or if a pair of braces is omitted, then only enough values from the (sub-) list are matched to fill the current sub-array. Therefore the previous declaration can be rewritten as:

```
static int keep[2][3][5] = { 1,0,0,0,
    0,2,3,0,0,0,4,5,6,0,0,7,8,9,10,0,
    11,12,13,14,15 };
```

Here the intermediate zero values must be present in order to preserve the intended order of matching.

Occasionally the reader may run into an initializer of the form:

= *constant-expression*;

as in:

```
static int block[5][5] = sizeof(int);
```

An initializer of this form only initializes the first element of the array to the indicated value; the rest are zero by default. Therefore **block[0][0]** would be assigned 2 on the reference system, and all others would be zero.

DEFAULT SIZE DECLARATION

If in a linear array declaration the size subscript is omitted or in a multi-dimensional array the leftmost size subscript is omitted, the compiler will supply this value by examining the initializer. For example, the declaration:

int prime[] = {1, 2, 3, 5, 7, 9, 0};

is equivalent to the declaration of **prime[]** presented previously. A declaration size of 7 is provided by the compiler.

AUTOMATIC ARRAY ELEMENT ASSIGNMENT

Because it is illegal to initialize automatic arrays, each element of an auto array must be assigned its value through a separate assignment statement. Figure 9.5 depicts an example of the assignment of values for the array in Program 9.2. Normal type conversion, as outlined in chapter two, will occur during assignment.

Notice that the array elements are handled much like individual char variables and that sequential manipulation is not necessary. Reassignment, including the use of scanf(), is perfectly valid and again is treated similarly to the corresponding fundamental data type variable. Particularly relevant is the use of the variable i as an index in lines 27 through 30.

Note that printf() utilizes the conversion specification associated with the array's declared data type (here int) when outputting individual element values. In addition, the %s specification will output the characters at the specified beginning address through the first zero (null) value as a character string. Strings will be discussed later in this chapter.

```
        /* Program 9.2 */
/* An example of the element assignment of an
   automatic array and the use of the %s
   conversion character.          */
                        /* 5 */
main()

{  int temp;
   char i, asgmt[4][2];
                        /* 10 */
   asgmt[0][0] =  ' ' ;
   asgmt[0][1] = '\63';
   asgmt[1][0] = 'Z';
   asgmt[1][1] = '\'';
   asgmt[2][0] = -350;      /* 15 */
   asgmt[2][1] = 110;
   asgmt[3][0] = .945e2;
   asgmt[3][1] = 0;
                        /* 19 */
   printf("\nInput alphanumeric symbol for ");
   printf("asgmt[2][1]:   ");
   scanf("%c",&asgmt[2][1]);
   printf("\nInput int value for asgmt[2][0]:\t");
   scanf("%d",&temp);
   asgmt[2][0] = temp;      /* 25 */
   printf("\n");
   for (i = 0; i < 8; ++i)
       { printf(" asgmt[0][%d] = %d \t",i,asgmt[0][i]);
         if ((i+1) % 3 == 0)
             printf("\n");  /* 30 */
       }
   printf("\nString asgmt[0][0] = %s",&asgmt[0][0]);
   printf("\nString asgmt[0][1] = %s",&asgmt[0][1]);
}
        /* 35 */
```

program output on following page

```
Input alphanumeric symbol for asgmt[2][1]:    @

Input int value for asgmt[2][0]:         63

 asgmt[0][0] =  0      asgmt[0][1] = 51     asgmt[0][2] = 90
 asgmt[0][3] = 39      asgmt[0][4] = 63     asgmt[0][5] = 64
 asgmt[0][6] = 94      asgmt[0][7] =  0
String asgmt[0][0] =
String asgmt[0][1] = 3Z'?@^
```

Figure 9.5. Program 9.2 demonstrates several common methods of manipulating array elements.

Array and Pointer Notation Equivalence

As mentioned earlier, a strong relationship exists between pointers and arrays in C. In fact, *an array name is a constant pointer in the C language.* This is why so few operations can be performed on entire arrays.

An array name represents the beginning address of a list of variables. An array name itself is not an lvalue. This contiguous series of related variables, or array elements, are true lvalues. However, unlike variables, these do not have individual names. They can only be referenced by the constant pointer value that is the entire array name. To this point, array elements have been referenced using array notation. For example, given the declaration:

long int lary[20];

The following expression would be used to reference this array's tenth element:

lary[9]

C compilers equate this with the expression:

*(lary + 9)

which resolves to an lvalue naming the contents of the tenth element of the array **lary**. The brackets can be interpreted as the indirection operation applied to the sum of a constant pointer (i.e. the array name) and the subscript. This explains why the following expressions are equivalent:

&lary[0]   ⇨   **&(*(lary +0))**   ⇨   **lary**

The most general form a bracketed expression can take is:

primary-expression[*expression*]

which is equivalent to:

*((primary-expression) + (expression))

The only difference between an array identifier and a pointer variable is that an array name is a constant pointer value with regards to any individual execution. It is illegal to attempt the assignment of a value to an array identifier.

Pointer Operations

A limited number of meaningful operations can be performed on pointers in C. The most common operation is the addition or subtraction of a constant value to a pointer value. By definition, an expression such as:

pointer-value + *integral-value*

is interpreted as the *integral value*th object (of the same data type as that pointed to by the pointer) past the stated *pointer-value*. Thus the C compiler implicitly factors in the length in bytes of each element. For example, assuming **long int**'s are

stored in four bytes, then the expression (**long_ptr + 6**) will result in a value twenty-four greater than **long_ptr**. Subtraction is executed in a similar manner.

One pointer value can also be subtracted from another. Both pointers must be to the same object data type. The result of this operation yields the int number of the number of objects that separate the two pointers. Reliable results are only guaranteed if the target objects are members of the same array.*

Pointer variables can naturally be assigned values, usually in association with the address of operator or an unadorned array name. Pointers are often explicitly assigned a value of zero because, in C, *a pointer value of zero is guaranteed to point to nothing*. Functions returning pointer values will inevitably return a zero value upon failure to perform successfully.

Comparisons of pointers with the equality and relational operators is also allowed. If this comparison is to be general and portable, the objects being pointed to must again be elements of the same array. Comparisons between pointers and non-pointer values are also permitted, but generally comparison with a value of zero is the only useful instance of this type of operation.

Aside from the associated "compound" operations, (i.e. incrementation and compound assignment), most compilers will disallow operations other than those previously outlined. Nevertheless, one can still perform meaningless operations, such as referencing elements beyond array bounds or comparing addresses of non-aggregated objects.

An Example Program

Program 9.3, presented in figure 9.6 illustrates the basic methodology of array and corresponding pointer use. It is

---

* More explicitly, this operation is guaranteed to yield relevant values if both operands are aligned at addresses that are multiples of the object length.

divided into three functions: st_fill( ) which fills the array with values from the standard input device, b_sort( ) which numerically sorts these values and orders them in the array, and main( ) which not only serves as a driver and I/O agent, but also duplicates the original array into the copy that is to be sorted (lines 18 and 19).

The correspondence of pointer notation and array notation is especially evident in: line 19 of main( ) and in st_fill( ) (pointer notation) versus b_sort( ) (array notation). With either notation, a copy of the pointer value of the beginning address of the specified array is passed to the called function. The called function may do whatever it wishes to with this value because it is only a local, temporary copy. But this copy can and is used to change the value of the target object (i.e. the array elements) through the process of indirection. These changes are real and binding.

The only code section that warrants explanation is the operation of b_sort( ). This function operates on a "bubble sort" principle. In this version each element is successively numerically compared with the next element until an element is found that is larger than the next one. Then three actions are performed (lines 11 through 15): the array positions of the two elements are switched; the index variable is set to point to the zeroeth element (line 14 plus the incrementation); and the process is started anew (line 15).

The alternative to pointer passed values is the use of array elements as actual arguments. This procedure has two possible shortcomings. It is inefficient in direct proportion to the number of array elements passed. Also the invoked function can only return one value which can then be used to uniquely change only one array element.

Finally, observe the use of preprocessor directives to include the necessary functions in file PROG93.C and to define the simple macro **SIZE** which increases the generality of this program.

```
/*      Program 9.3 consists of 3 functions
   organized in 3 files: main() in PROG93.C ,
   b_sort() in B_SORT.C, and st_fill() in
   ST_FILL.C.  There are numerous illustra-
   tions of the manipulation of arrays by
   both pointer and array notations      */

#include "B_SORT.C"
#include "ST_FILL.C"
#define SIZE 10
                    /* 11 */
main()

{  int ary[SIZE], orig[SIZE], index;
   char ans;              /* 15 */
   st_fill(orig, SIZE);
       /* copy ary[] from orig[] */
   for (index = 0; index < SIZE; ++index)
       *(ary + index) = orig[index];
                          /* 20 */
   b_sort(&ary[0], SIZE);
   printf("\nDo you want the unsorted vs.");
   printf("\nthe sorted version output?- ");
   scanf("%1s", &ans);    /* 24 */
   if ( ans == 'y' || ans == 'Y')
       { printf("\n    sorted    original");
         for ( index = 0; index < SIZE; ++index)
             printf("\n  %5d    %5d ",ary[index],
                                orig[index]);
       }                  /* 30 */
}


          /* file ST_FILL.C */
/* inputs int values from standard input
   to fill an int array of size max    */

st_fill(p_ary, max)
int *p_ary, max;
                   /* 7 */
```

```
(   int i;
    printf("\n input integer values");
    printf(" for array \n");   /* 10 */
    for (i = 0; i < max; ++i)
        ( printf("   array[%d] :   ",i);
          scanf("%d", (p_ary + i) );
          /* or scanf("%d", &p_ary[i]) ); */
        )                            /* 15 */
)

          /* file B_SORT.C */
/* bubble sorts an int. array of size
   max, returns nothing           */

b_sort(sort, max)
int sort[], max;
              /* 7 */
( int i, hold;
   for (i = 0; i < (max - 1); ++i)
       if (sort[i] > sort[i + 1])
           ( hold = sort[i];
             sort[i] = sort[i + 1];
             sort[i + 1] = hold;
             i = -1;
             continue;  /* 15 */
           )
)
```

```
    input integer values for array
      array[0] :  91
      array[1] :  17
      array[2] :  -3
      array[3] :  2456
      array[4] :  57
      array[5] :  00
      array[6] :  9
      array[7] :  0
      array[8] :  -3
      array[9] :  69
```

```
Do you want the unsorted vs.
the sorted version output?- Y

    sorted       original
     -3            91
     -3            17
      0            -3
      0           2456
      9            57
     17             0
     57             9
     69             0
     91            -3
   2456            69
```

Figure 9.6. Example of the manipulation of array elements and addresses.

Character Strings

A special subset of arrays in C is the string data type. Although this text has utilized strings as the initial format strings for the printf() and scanf() functions, they have not been formally defined.

A string is a sequence of characters of arbitrary length surrounded by double quotes. The delimiting double quotes are not part of the string, but serve only as punctuation. (An actual double quote character can be realized by the escape sequences \" or \ 46 in ASCII). Likewise other escape sequences may be used as individual elements of a string. The programmer need not associate a length with each string, rather a special character is used to demarcate the left boundary of the string. The C compiler automatically places this end character, which is the null character (i.e. ' \ 000', " or numeric zero) at the

end of each string. This string terminating character is the last element of all strings, and any string generating section of code must have a procedure to place a null character at the end position. Because of its special purpose, the null character cannot be used as a non-terminating element of a string.

For example, the string:

$$" \backslash nString \backslash 72 \backslash t"$$

would be internally represented by the following sequence in main memory:

address X	x=1	x=2	x=3	x=4	x=5	x=6	x=7	x=8	x=9
\n	S	t	r	i	n	g	\72	\t	\0

Of course each character would be converted into its encoded equivalent.

Every occurrence of a string constant in code generates a pointer to the first element of the string. This is consistent with the manner in which array objects are treated. When a string is used as a actual argument member, the corresponding formal argument must be a character array or character pointer variable.

From the previous discussion, two methods of creating a "string variable" suggest themselves. The first utilizes a pointer to a char variable, and is exemplified by the two lines:

```
char *headm;
headm = "Dear sir\n";
```

which can optionally be combined into one statement:

```
char *headm = "Dear sir\n";
```

The second equivalent method utilizes array notation:

```
char headm[] = {'D' 'e', 'a', 'r', ' ', 's', 'i', 'r', '\n', '\o'};
```

Character array initializations can, in fact, make use of string notation, as in:

char headm[] = **"Dear sir \n";**

The array size for the last two declarations has been supplied by the compiler as ten.

After either type of declaration, the pointer to character string variables can be manipulated as for a normal array. The presence of the null terminator often introduces shortcuts and conveniences not associated with normal arrays. Figure 9.7 presents two equivalent versions of a function for comparing the value of two strings. In the comparison, corresponding characters are sequentially compared one at a time until a character value does not match. The string with the lower first different character value is the "lower" string. The zero value of the null character ensures that shortened words will be evaluated as lower in value than their parents; thus **dog** is less than **doggy**. A version of this function is often found in the standard C library. Note again the correspondence of pointer and array notation.

There is a special conversion specification for outputting a character string through printf() — %s. Refer back to Program 9.2 in figure 9.5. Note how the char array **asgmt[]** was seemingly output by line 33. But since the first character in the array is a null character, the string starting at **asgmt[0][0]** is only one character long and is the *null string*. The null string can be represented by "", " \0", or **null[1]** = {0}. The second string is output starting at as **asgmt[0][1]** (line 34). Because the array ends with the null value, the second through the last elements are output as characters. Through the use of pointers or array subscripting it is possible to reference conceptual sub-arrays as was done with this second printf().

```
           /* Program 9.4 */
/* compares 2 strings char by char, as in
   alphabetical indexing and returns one
   of the following values:
           0: two strings are equal
           1: 1st string is greater than 2nd
           -1: 2nd string is greater than 1st   */

/*   ********** pointer version **********  */
strcmp(r,s)
char *r, *s;
                    r      s
{  for ( ; *r == *s; ++r, ++s)
        if (*r == 0)
           return(0);
   if (*r > *s)
       return(1);
   else
       return(-1);
}


/*   ********** array version **********  */

strcmp(r,s)
char r[], s[];

{  int i;
   for (i = 0; r[i] == s[i]; ++i)
        if (r[i] == 0)
           return(0);
   if (r[i] > s[i])
       return(1);
   else
       return(-1);
}
```

Figure 9.7. Two versions of the standard C library
function strcmp(), demonstrating equivalence
of pointer and array notation

The conversion specification %1s is often used in scanf( ) specification strings. It has the same effect as %c but skips all intervening whitespace (see Program 9.3, main( ) line 24).

The pointer / null character combination makes certain string manipulations very easy. For example, the effective length of a string can be easily changed by moving the occurrence of the null terminator.

Arrays of Pointers

Pointer values can be stored in array format much like any fundamental data values. For example, an array of ten pointers to functions returning double's could be declared through a statement such as:

```
double (*pf[10])();
```

Then assuming that the function log( ) has been previously defined or redeclared, an element of this 10 x 1 array can be assigned its address through a statement such as:

```
pf[9] = log;
```

An explanation of complex declarations such as this one is given in appendix F.

Often it is advantageous to reference a number of strings through an array format. One method of accomplishing this is by actually creating an array whose elements are the desired characters of the string. For example, this method is followed in the initialization of the array **gulf_st**, as shown in figure 9.8a. This array represents a memory block sixty bytes long, composed of five twelve-byte long blocks each containing the name of a state. The disadvantage to this scheme derives from the fact that subarrays must be of equal length. For example, the subarray containing Florida has four trailing char elements that are initialized to zero by default. Only one of these is

required as a null terminator.

Array to pointer notation conversion is accomplished with the following steps:

- The multiplication and addition to mimic the index notation.
- Factoring this value by the length of the object pointed to.
- Then adding the beginning address of the array.

For example, **gulf[3][7]** would be expanded by the compiler to:

*(gulf + (((3 * 12) + 7) * 1))

Another method of grouping strings is by the formation of an array of pointers to char (strings). Since an occurrence of a string is actually converted to a pointer to that string, this format is very natural. Now consider figure 9.8b. The array **pgulf_st** is initialized to pointer values to the five specified strings. The strings themselves are located in memory as the operating system sees fit. Because strings are handled through pointers, the elements act effectively as their respective strings themselves. For example, the following two lines assign the value of (the pointer to) "Mississippi" to the (pointer to a) string variable **state**:

char *state;
state = pgulf_st[3];

```
a)  static char gulf_st[5][12] = {
        { 'F','l','o','r','i','d','a' },
        { 'A','l','a','b','a','m','a','\0' },
        { 'M','i','s','s','i','s','s','i'
                        ,'p','p','i' },
        { 'L','o','u','i','s','i','a','n','a' },
        { 'T','e','x','a','s','\0','\0','\0' }
                                }
b)          static char *pgulf_st[5] = {
                    "Florida",
                    "Alabama",
                    "Mississippi",
                    "Louisiana",
                    "Texas"
                                }
```

Figure 9.8. Examples of two methods of grouping strings:
a) an array of characters   b) an array of pointers
to the character strings.

An array of pointers to strings has the advantage that the referenced strings can be of different lengths. Referencing of pointer values of the array is again accomplished through an indirection operation.

Pointer arrays can naturally be used to point to any valid type of object, not just to strings. Objects of different lengths can also be more strongly grouped together by their inclusion in a member, which is the subject of the next chapter.

10

Structures, Unions, and Bit Fields

Introduction

In the last chapter, the array derived data type was introduced. Arrays are the preferred method of storing objects of the same data type. In addition to arrays, C includes three more derived data types that can be used to store information: structures, unions, and bit fields. Unlike arrays, these can be used to store data of the same type or data of differing types.

413

Each individual data item within a structure, union, or bit field is referred to as a *member*. In most instances, an operation can only be used with an individual member as an operand. However, certain operators allow the entire structure, union, or bit field itself to be used as an operand.

The majority of this chapter is devoted to the structure data type because it is the most widely employed. This discussion of structures will help you understand unions and bit fields, however, because many of the attributes of structures are directly applicable to unions and bit fields.

STRUCTURES

A structure is a method of grouping logically related data items which are stored in memory. A structure is referenced in a C program by a structure variable. The concept of a structure is analogous to that of a record in many other languages.

Two central attributes differentiate a structure from an array:

- Its members need not be of the same data type.
- A structure variable is an actual lvalue while an array name is a constant pointer to the beginning of a series of similar data items.

Declaration and Initialization

A structure is declared in much the same manner as an enumerated data type. For both, a template is used to define a particular data type. Once the data type has been defined, individual structure variables may then be declared to be of that type.

An employee record will be used to illustrate the process of template definition and structure variable declaration. Suppose each record was to contain the following information: name, social security number, pay rate, and a weekly output

rating between one and ten. The corresponding structure template in C would have the form:

```
struct emp_rec {
    char *name;
    long ss_num;
    float payrate;
    int wk_rtg[57];
                }
```

The keyword **struct** specifies a structure data type definition. It is followed by an identifier (here **emp_rec**), called a *structure tag*, which names this particular user-defined structure data type. The structure members are ordinally declared within braces. A structure tag definition does not cause the allocation of memory in storage, rather it serves as a guide to the compiler for future declarations of structure variables of the corresponding type.

For example, the following statement declares the variables **sales, prod,** and **prsnl** to be structures of emp_rec type:

```
struct emp_rec   sales, prod, prsnl;
```

In this manner each variable is declared an aggregate data type consisting ordinally of the following members: a character pointer **name**, a long integer **ss_num**, an unsigned integer **payrate**, and a 57 x 1 array of integers named **wk_rtg**.

Generally, identifiers such as the structure tag, structure members, and structure variables must follow the syntax rules outlined in chapter 2. The most relevant of these syntax rules is that variable identifiers must be unique.

EXCEPTIONS TO IDENTIFIER UNIQUENESS

There are three exceptions to structure identifier uniqueness. The first allows the repetition of a structure variable name

within program sublevels. This exception follows the same logic as the analogous situation which occurs with fundamental data type identifiers, which is covered in chapter 7.

Secondly, a structure tag or member name can duplicate a non-structure variable name without conflict because contextual clues differentiate the two uses to the C compiler. For example, given the previous definition of emp_rec, the following line should not produce an error message:

```
int emp_rec, ss_num;
```

The final exception pertains to situations where two or more structures share the same scope. These structures can share identical member names as long as the data types are also identical. However each consecutive member name and corresponding data type that is shared by these structures must be identical. Once differing member names and/or data types are encountered, subsequent sharing of identifiers is not allowed. The compiler will determine to what structure variable each member identifier occurrence refers to.

INITIALIZATION OF STRUCTURES

Like arrays, only extern and static structures can be initialized. This is accomplished by postfixing a structure variable with an assignment operator and the appropriate brace enclosed list of literals. For example, if the previous template was externally defined, the declaration:

```
struct emp_rec  dvlpmt = { "LaRosa, M",
        526219488, 2050, {0,8,9,11} };
```

would intialize: **name** of **dvlpmt** to a pointer to the string "M.LaRosa"; **ss_num** of **dvlpmt** to **526219488**; **payrate** of **dvlpmt** to **2050**; and **wk_rtg[0]** to **0**, **wkrtg[1]** to **8** etc. Inner brace pairs are commonly used to group initializer lists for

members that are themselves aggregate objects such as arrays or structures. Initialization of these objects follow the rules for their non-member counterparts. (Consequently the elements **wk_rtg[4]** through **wk_rtg[57]** are default initialized to zero.) Trailing members that are unmatched to initializers are implicitly initialized to zero.

VARIATIONS ON THE THEME

A template can be defined and structure variables declared in the same statement. For example, the statement:

```
struct calndr {
    int month;
    int day;
    } yr_2000, yr_2001;
```

defines structure variables **yr_2000** and **yr_2001** to be of type calndr. Initialization of static or extern structure variables can also occur in the same declaration; as in:

```
struct calndr {
    int month;
    int day;
    } yr_2000 = { 12, 30 },
      yr_2001 = { 1, 1 };
```

This statement accomplishes the same objectives as the previous statement, and in addition initializes **month** of **yr_2000** to **12**, **day** of **yr_2000** to **30**, etc.

The structure tag may be omitted from a definition. When the tag is absent, only those structure variables listed at the end of that definition can be declared to be of that structure type.

Internal Storage of Structures

Structure members are stored in memory in the same

order in which they are declared. However, since structures are often composed of members of different data types, and since different data types often have different address alignment requirements, there may be unused, slack bytes inserted between members to preserve correct member alignment. (The concept of alignment is covered in chapter 3's section "Numeric Representation".) Alignment requirements are machine dependent.

In order to demonstrate the concept of slack byte insertion, consider the structure declaration depicted in figure 10.1a. Assume that int's require alignment on a double byte boundary (i.e. those addresses evenly divisible by two), while char's can be placed at any byte. The structure variable **example** would then occupy memory as shown in figure 10.1b. The first integer **n** is placed after an even address (i.e. 2x) as required. The character **c** follows immediately, as it has no alignment demands (other than occupying a complete byte). The second integer **o** cannot then immediately follow **c** because the next location is not an even byte as int alignment requirements demand. Therefore a slack byte is inserted to align **o** as required.

a.
```
struct align {
      int n;
      char c;
      int o;
      } example = { 10, '?', 500 };
```

b.

member address	2x+0	2x+1	2x+2	2x+3	2x+4	2x+5
	int n = 10		char c = '?'	slack byte	int o = 500	
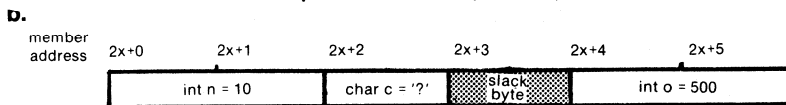
Figure 10.1. a) Example structure declaration with inherent alignment conflict.

b) Resolution of alignment conflict through insertion of slack byte.

Operations on Structures

There are only a few operations that can be performed on an entire structure. Orginally only two operations were allowed. The address of (the beginning of) a structure can be taken by prefixing the structure variable name with the "address of" operator, **&**. Access to a structure member can be obtained by postfixing the structure variable name with the member access (dot) operator followed by the name of the appropriate member. The following two expressions illustrate these operations on an earlier example variable:

```
&sales
sales.ss_num
```

Since the inception of C, two additional operations on structures have been added. The first allows assignment of one structure's variable values to another structure of the same user defined type. For example, one emp_rec flavored structure can have its members assigned to another through a statement such as:

```
prod = sales;
```

Also a copy of an entire structure variable can be passed to or from a function by inclusion of the variable in the actual argument list. The corresponding formal argument should be of the same data type as the actual argument. Care must be taken as some older compilers treat, without warning, an unadorned (i.e. no **&** operator prefix) structure variable name as a pointer to that structure. For further discussion on structures, see the section entitled "Structures and Functions" in this chapter.

As was true for separate array elements, separate structure members can be involved in any operations that their individ-

ual data type allows.

An Example Program

Figure 10.2 contains a very simple program utilizing structures. Program 10.1 makes use of a slightly altered version of the employee record. The program calculates the week's pay of a given employee. It includes calculations for normal hourly and "time and a half" overtime earnings (lines 26 through 32). Also employees can earn a bonus incentive, which is calculated as 1% of total gross wages for every rating point over 7 (lines 33 and 34).

With respect to this chapter, the main point to note is that members can be manipulated exactly like their non-member counterparts. Every direct reference to a member variable must, however, be preceded by the structure variable name and the dot operator.

```
/*        Program 10.1 - illustrates manipu-
   lation of structure member (variables).   */
main()
                                  /* 4 */
{  float ovrtm = 0.0, bpay = 0.0, npay;
   struct wk_pay  {
        char *name;
        long ss_num;
        float payrate;
        int wk_rtg;
        float hpw, tpay;
                 } input;
                                  /* 13 */
   printf("\nemployee's last name: ");
   scanf(" %s ", input.name);
   printf("social security number ");
   printf("\nwithout hyphens: ");
   scanf(" %ld", &input.ss_num);
```

```
printf("this week's rating: ");
scanf(" %d", &input.wk_rtg);
printf("pay per hour (eg $4.55): ");
scanf(" $%f ", &input.payrate);
printf("hours worked this week (eg 38.66): ");
scanf(" %f ", &input.hpw);
                                 /* 25 */
if (input.hpw > 40.0)
   { ovrtm = input.hpw - 40.0;
     npay = ovrtm * (input.payrate * 1.5) +
                40 * input.payrate;
   }
else
   npay = input.hpw * input.payrate;
if (input.wk_rtg > 7 && input.wk_rtg <= 10)
   bpay = (input.wk_rtg - 7) * npay / 100;
input.tpay = npay + bpay;
                                 /* 36 */
printf("\n %s   ss#: %d ", input.name,
                        input.ss_num);
printf("\n hourly pay: %.2f   bonus pay: $%.2f"
                  ,npay, bpay);
printf("\n\t total pay: %.2f", input.tpay);
}
```

```
employee's last name:   Jenkins
social security number
without hyphens: 123456788
this week's rating: 9
pay per hour (eg $4.55): $5.00
hours worked per week (eg 38.66): 45.00

 Jenkins    ss#: 123456788
 hourly pay: 237.50    bonus pay: 4.75
         total pay: $242.25
```

Figure 10.2.   Program for calculating a week's gross pay of an
employee based on hourly wage, overtime,
and performance bonus.

Structures and Functions

There are four methods of transferring the values of a structure from one function to another. Three of these make use of the function call. The most elementary method involves supplying each structure member as an actual argument of the function call as shown below:

```
main()

{   .   .   .
    hyp(input.name, input.ss_num,
          input.payrate, input.wk_rtg);
    .   .   .
}
```

These actual arguments are then treated as separate, non-structure values (unless they are themselves structures), and the formal arguments need not be structure members. The called function which corresponds to the example shown above could be:

```
hyp(fname, fss_num, fpayrate, fwk_rtg)
char *fname;
long fss_num;
float fpayrate;
int fwk_rtg;

{   .   .   .
}
```

The value held by: **name** of **input** would be passed to **fname, ss_num** of **input** would be passed to **fss_num**, etc.

This is obviously the preferred practice when only a few members' values need to be passed. It is also acceptable when an entire small structure's values need to be passed. With larger structures this practice quickly becomes untenable by reason of

its coding and execution inefficiency.

The second method for transferring the values of a structure from one function to another involves sending a copy of an entire structure to the called function. This is illustrated in Program 10.2a of figure 10.3. This program reveals three main points concerning structure passing:

- A structure variable name alone represents the whole structure and not a pointer to the structure. A structure name is a valid formal or actual argument under compilers supporting the aforementioned extensions.

- The structure variable used as an actual argument in the calling function and its matched formal argument in the called function must be of an identical structure data type.

- Auxiliary functions must be declared in a manner appropriate to the data object they return. For example in line 31, the function val_pln() is declared as a function returning a copy of a structure variable of type values.

- If the called function is positioned lower in the source file then the calling function, then the called function must be redeclared in the calling function. In Program 10.2.a, since the called function val_pln() is positioned lower in the source file, it is redeclared on line 12.

- An entire returned structure can be copied into a structure of identical type (line 20). In Program 10.2a, a copy of the structure variable **str** is returned from line 38. It substitutes for the function call in line 20 and is subsequently copied into the structure variable **base**. Note that both **base** and **str** are declared as the type values.

```
/*      Program 10.2a - illustrates the
   method of passing entire copies of
   structures to an auxiliary function.*/
                          /* 4 */
struct values {
    int val1;
    long val2;
    float val3;
              };
main()
                          /* 11 */
{  struct values val_pln();
   int n;
   struct values  base = { 10, 50000, 5.5 };
   printf("\nWhat int value do you want ");
   printf("to increment \neach structure ");
   printf(" member by ?:   ");
   scanf(" %d ", &n);        /* 18 */

   base = val_pln(base, n);

   printf("\n The incremented values are:");
   printf("\n   val1 = %d",base.val1);
   printf("\n   val2 = %ld",base.val2);
   printf("\n   val3 = %.2f",base.val3);
}                          /* 26 */

/* val_pln() increments each member of
   passed copy of structure by value n,
   then returns changed structure      */
struct values val_pln(str, incr)
struct values  str;         /* 32 */
int incr;

{  str.val1 += incr;
   str.val2 += incr;
   str.val3 += incr;
   return(str);
}                          /* 39 */
```

program output on following page

```
What int value do you want to increment
each structure member by ?:   7

 The incremented values are:
    val1 = 17
    val2 = 50007
    val3 = 13.60
```

Figure 10.3.   Program demonstrating the use of passed copies
of entire structures.

Note in Program 10.2a that the declarations are consistent throughout because an external definition of **values** was used in lines 5 through 9. Both argument structures and the structure that is returned are declared to be of this type.

There are two drawbacks to this approach. First, many compilers at this time do not feature the language extensions that allow structure passing. Additionally, passing whole structures is considerably more time and memory inefficient than utilizing the third approach, which deals with pointers.

As was stated, the starting address or the constant pointer value to a structure variable is obtained by prefixing that structure name with the unary **&** operator. For example, the structure **base** from Program 10.2a can be addressed through the expression:

&base

This pointer value can be readily passed to a called function and should be matched onto a parameter declared in the appropriate manner. The declaration:

struct values *psv;

states that **psv** is a pointer variable to a structure of type values. Notice that **psv** is declared as a pointer to the type of structure and not to the particular structure **base** itself.

In the called function, the entire structure **base** can then be referenced indirectly through the use of the unary operator ***** applied to the pointer variable, as in the expression:

*psv

Typically, separate members are manipulated in the called function. Since the previous expression is indirectly equivalent to **base**, a member of this structure, for instance **val1**, could be referenced by the expression:

(*psv).val1

where the associative parentheses are required to bind the unary indirection operator to the pointer variable in preference over the primary dot operator. Because of the demand for expressions of this last type, C has provided the special compact operator, the primary -> operator. The most recent expression is exactly equivalent to:

psv->val1

Therefore the -> operator substitutes for the **(*).** tokens. The -> operator is known as the *member selection, member pointer,* or *arrow operator.* Both the member access and member selection operators have a left to right associativity and the highest possible precedences.

A version of Program 10.2, altered to manipulate structure members via a pointer, is presented in figure 10.4. The more prominent changes include:

- There is no redeclaration within main() of the auxiliary function because it now does not return a value.

This called function manipulates the structure **base** indirectly through the pointer **psv**.

- No assignment is necessary in the function call of line 20. The auxiliary function changes **base**'s member values through indirection.

- The pointer variable **psv** is passed the starting address of structure variable **base** during the function call (line 20). Hence **psv** must be declared as "a pointer to a structure of type values". For this reason the values template definition must be external (i.e. so as to be accessible to the auxiliary function).

```
/*      Program 10.2b - version of 10.2a
   that uses passed pointer values to
   indirectly reference members.        */
                                /* 4 */
   static struct values {
       int val1;
       long val2;
       float val3;
                              };
main()
                            /* 11 */
{  int n;
   static struct values  base = { 10,
                50000, 6.6 };
   printf("\nWhat int value do you want ");
   printf("to increment \neach structure");
   printf(" member by ?:   ");
   scanf(" %d ", &n);        /* 18 */

   val_pln(&base, n);

   printf("\n The incremented values are:");
   printf("\n    val1 = %d",base.val1);
   printf("\n    val2 = %ld",base.val2);
   printf("\n    val3 = %.2f",base.val3);
}                            /* 26 */
```

```
/* val_pln() increments each member of
    original structure by n, through the
    process of indirection                */
val_pln(psv, incr)
struct values  *psv;            /* 32 */
int incr;

{  psv->val1 += incr;
   psv->val2 += incr;
   psv->val3 += incr;
}                               /* 38 */
```

Figure 10.4. Version of program 10.2a designed to utilize a structure pointer and indirect accessing of structure members.

There is a fourth method of accessing structures between functions. It is not, however, associated with the function call itself. Externally declared structure variables are available to functions in the same manner as external fundamental type variables (chapter 7).

Templates have a storage class associated with them. Externally defined templates should occur before all the functions in a file as some compilers will not allow a reference to a template defined later in the file (i.e. a "foward reference").

Structures and Arrays and Pointers

The two aggregate variables in C — arrays and structures are often combined in a number of ways to create useful derived data types. Some of the most common and simple of these combinations are described next.

STRUCTURES CONTAINING ARRAYS

Consider the following declaration:

```
struct day (
    int month;
    char *d_name[8];
    int date[32];
            ) july;
```

The members of this structure can be handled much like their non-structure counterparts except that each must be preceded by the structure variable name and the dot operator. For example, the following expressions access the first element of **d_name[ ]** and the last of **date[ ]**, respectively:

```
july.d_name[0]
july.date[7]
```

Because the precedence of the dot operator is so high, member access expressions are normally taken as a unit. However, associative parentheses can always be placed around the entire expression for security.

ARRAYS OF STRUCTURES

A higher derived data type of this sort is very commonly used because a large number of similar records are often processed together. A review of the structure type emp_rec shows that structure variables of this type, such as **sales, prod,** and **prsnl**, could only hold one set of values at any one given time. Yet unless the described company was very small, there would be more than one person employed in each department. The logical solution to this problem is the declaration of an array of emp_rec structures, as shown next:

```
struct emp_rec {
    char *name;
    long ss_num;
    float payrate;
    int wk.rtg;
    } sales[8], prod[40], prsnl[4];
```

This statement declares **sales[ ]** to be an array of eight blocks of memory, with each block containing a structure of type emp_rec.* The variables **prod[]** and **prsnl[]** are likewise declared to be arrays of forty and four emp_rec flavored structures respectively. The arrays could have also been declared after the template definition as through:

```
struct emp_rec   sales[8], prod[40], prsnl[4];
```

External and static arrays of structures can be initialized much as one would expect. It is often worthwhile to further brace off each element's values. For example, the array **prsnl[ ]** could be declared and initialized (after the template definition) as follows:

```
static struct emp_rec   prsnl = {
    { "Scoulos M",638214800,6.55,3 }
    { "Eliot T",238944204,4.05,2 }
    { "Stepien D",841348430,9.00,9 }
    { "Snorbie V",333445551,7.23,5 } };
```

To access a member of this structure, the array name and bracketed element subscript must be supplied before the dot operator. For example, the pay rate of Mr. Stepien can be referenced through the expression:

---

* The structures comprising the array may or may not be contiguous. Slack bytes will be inserted between each constituent structure as demanded by the alignment requirement of the first member of each structure.

```
prsnl[2].payrate
```

A pointer value to this array of structures can be declared through a statement such as:

```
struct emp_rec *p_er;
```

while the statement,

```
p_er = &prsnl[0];
```

assigns the beginning address of the entire array of structures, which is also the beginning address of the first structure, to **p_er**. The expression

```
p_er->payrate
```

indirectly references the third member of the first structure element, which was initialized as **6.55**. Incrementing the pointer by one causes **p_er** to point to the beginning address of the second structure element of array **prsnl[]**. The expression **++p_er->payrate** increments **payrate** of the current structure because the arrow operator has a higher precedence than the increment operator. On the other hand, the expression **(++p_er)->payrate** increments the pointer, then references the pay rate of the next structure in the array.

STRUCTURES CONTAINING STRUCTURES

A structure can itself contain one or more structures as members. For example, if the emp_rec template was previously defined and currently active, the following structure template definition would be valid:

```
struct tcomp {
    struct emp_rec   cur_wk;
    struct emp_rec   last_wk;
    float ratio;
              };
```

The data type tcomp is a structure consisting of two structures of type emp_rec and a floating point quantity. The majority of compilers will not allow the definition of one template (e.g. emp_rec) inside another (e.g. tcomp). However, other previously described initialization and alternative declaration schemes can be used.

Members of the inner structures can be referenced through a process of chaining of the dot and arrow operators. Given the declarations:

```
struct tcomp   cprsnl;
struct tcomp   *p_tc = &cprsnl;
struct emp_rec   *p_er = &cur_wk;
```

the following expressions:

```
cprsnl.cur_wk.wk_rtg
p_tc->p_er->wk_rtg
```

both reference the integer quantity **wk_rtg** which is a member of the structure **cur_wk**, which in turn is a member of the structure **cprsnl**. Both the dot and the arrow operator have a right to left associativity. Note once again that operations are performed on the structure variables and not on the tags **tcomp** and **emp_rec**.

POINTERS WITHIN STRUCTURES [Advanced]

It should not be suprising at this point to learn that pointer variables can be members of a structure. Consider the statement:

```
struct ll_val {
    struct ll_val  *bp;
    int val1, val2, val3, val4, val5;
    int tot, *index;
    struct ll_val  *fp;
            } s1, s2, s3 ;
```

Two pointer variables of type "pointer to structure of type ll_val" and one of the type "pointer to integer" are defined. The former two declarations are said to be *self-referential* in nature, since they are defined to point at structures of the type they themselves occur in.

The self-referential pointers can be used to form a *logical linked list*, which is a set of data types (here structures) that are conceptually grouped together in a linear fashion through the use of pointers. For example, the structure declaration:

```
struct ll_val   s5 = { &s1,
        5,7,11,13,17,53,
        &s5.val1, &s3 };
```

sets s5's pointers as follows: **bp** points to structure variable **s1**, **fp** to s3, and **index** to **val1** of the same structure. If we interpret **bp** and **fp** as backward and forward pointers, then a logical list in the order s1 → s3 → s3 is established. The logical order here does not coincide with the physical order of the structures in memory. The **fp** and **bp** variables in the other structures s1 through s3 could likewise be assigned consistent values, thus forming a complete list. Then the next structure of such a list could be addressed as through the expression:

```
s5.fp
```

which yields the address of s3, while the expression:

```
s5.fp->val3
```

yields the value of **val3** of **s3**. Assuming that all the pointers have been set to acceptable values, then expressions of the sort:

s5.bp->bp-> . . . ->val3

can be used to reference values that are several places removed on the list. The pointer **index** can be set to any active integer variable, whether a member of one of these structures or not.

Many high order logical data groupings such as trees, other sorts of chains or lists, tables, etc — are commonly utilized in programming. The discussion of these topics is beyond the scope of this book, however, it is worthwhile to note that these logical data groupings make heavy use of arrays and structures containing pointers.

UNIONS

One of the more unique derived data types in C is the *union*. The union data type is similar to the structure data type in that both can contain members of different types and sizes. However, a union variable can hold, at any one time, only one of its possible members. On these grounds the union data type can be viewed as a sort of user-defined, general purpose, non-aggregate data type.

Declaration and Initialization

The declaration of a union closely parallels that of a structure except the keyword **union** is utilized. Explicit initialization of unions is not allowed on some compilers. On others only the first member of static or extern unions may be explicitly initialized. All the members of an external or static union are implicitly initialized to zero by default. For example, the statement:

```
static union fund {
    char c;
    int i;
    float f;
    double d;
            } x, y, z;
```

declares **x, y,** and **z** to be unions of type fund. Each of their members, **c, i, f,** and **d** are default initialized to zero. Except for explicit initialization, the same shorthand occurs for unions as for structure declarations.

Internal Representation of Unions

From a memory management standpoint, a union can be thought of as a structure with its members offset by a zero amount. That is, all the members of a union begin at the same address, which is also the beginning address for the entire union. The members, or at least the beginning portions of all the members, are stored in the same location — basically on top of each other. This explains why only one member can be held at a time: any data from other members subsequently stored is written on top of (all or part of) the current member. The data that is over-written is of course lost. The compiler assigns a union a block of memory that satisfies alignment and length requirements for all of the member data objects.

Applications of Unions

The same operations can be performed on unions as on structures. The members of a union are accessed directly through an expression of the form:

union-name . member-name

or directly through:

union-pointer - > member-name

where the pointer is declared as the proper user defined type.

Program 10.3, presented in figure 10.5, depicts the use of the union data type.

```
/*      Program 10.3 - example of nature and
    use of a user-defined union data type      */
main()

{   int i;                          /* 5 */
    static union example (
        int i;
        float f;
        int ary[4];
                        } x;        /* 10 */

    printf("Input four integers: \n");
    for (i = 0; i < 4; i += 2)
        scanf("%d %d", &x.ary[i], &x.ary[i+1]);
    x.i = 1321 * 4;                 /* 15 */
    printf("\t\t x.i = %d", x.i);
    x.f = x.i * .56;
    printf("\n  x.f = %.2f  x.i = %d\n", x.f, x.i);
    for (i = 0; i < 4; ++i)
        { printf(" ary[%d] = %5d", i, x.ary[i]);
          printf((i == 1) ? "\n" : "    ");
        }                           /* 22 */
}
```

```
Input four integers:
 21 133  2547 51
                   x.i = 5284
  x.f = 2959.04  x.i = -3933
 ary[0] = -3933    ary[1] = 17720
 ary[2] =  2547    ary[3] =    51
```

Figure 10.5.   Program demonstrating the nature of the union data type.

The union type **example** and the variable **x** of this type are declared on lines 6 through 10. Since this is a static union, all members are initialized to zero by default. Lines 12 through 22 assign values to the union members in a variety of ways and then outputs these values. The main point to note is that when a different member is assigned a new value (lines 15 and 17), the new value partly or completely supercedes the previous member's value. Therefore, only a single member value can be faithfully stored in a union at any one time. It is up to the programmer to insure that the value being accessed represents the intended data type and member variable. For example, the reference to **x.1** in line 18 is inappropriate since the first half of a **float** value has replaced the former integer value by the assignment in line 17.

Unions are more rarely used than structures, and in any event, unions are in no sense a vital data type. The union data type was included in C mainly as a convenience, although there are a few instances that strongly suggest its use.

BIT FIELDS

At a systems level, a programmer is sometimes faced with the need to manipulate not only separate bytes of data, but also

individual bits. Bit manipulation, or "bit-flipping" as it is lightly referred to, can be accomplished using only the data types introduced up to this point and the shift and the bitwise logical operators introduced in chapter 4. Despite this fact, C also has included a more direct method of accessing small bit fields —namely through the inclusion of bit field variables in structures. Bit fields allow the direct manipulation of a string of preselected bits as if the string represented an unsigned integral quantity.

Declaration and Initialization

The declaration of a bit field variable within a structure takes the form:

unsigned *identifier* **:** *bit-length*;

Where the bit length is the number of bits associated with this identifier. This integral number may not exceed the length of an int in bits, which is a machine dependent quantity. Some compilers support other types of bit fields; on many an int field declaration is automatically converted to unsigned.

In addition to bit field variables, padding bit fields can also be declared. These fields perform the same purpose as slack bytes within structures, namely that of alignment of the next object at a predetermined point. Padding fields are unnamed, and their declarations have the following form:

unsigned   **:** *bit-length*;

A bit length of zero specifies the particular padding length necessary to align the next object on an int boundary.

All other aspects of the declaration and initialization of the associated templates and structure variables parallel that of normal structures.

Internal Representation of Bit Fields

The internal representation of bit fields is machine dependent for two reasons: the int data type memory allocation differs among machines, and while the majority of machines store fields ordinally left to right, some store them right to left in memory. For this discussion, a two byte int length and a left to right storage format will be assumed.

Figure 10.6 contains a bit field declaration and the attendant memory representation. Aside from the general form of the declaration and the ordinal matching of bit field variables to initializers, there are several specific points to observe.

a)
```
static struct fill {
     unsigned f1 : 1;
     unsigned f2 : 2;
     unsigned f3 : 4;
     unsigned    : 1;
     unsigned f4 : 6;
     } b_int = { 1,0,15,1,33 };
```

b)
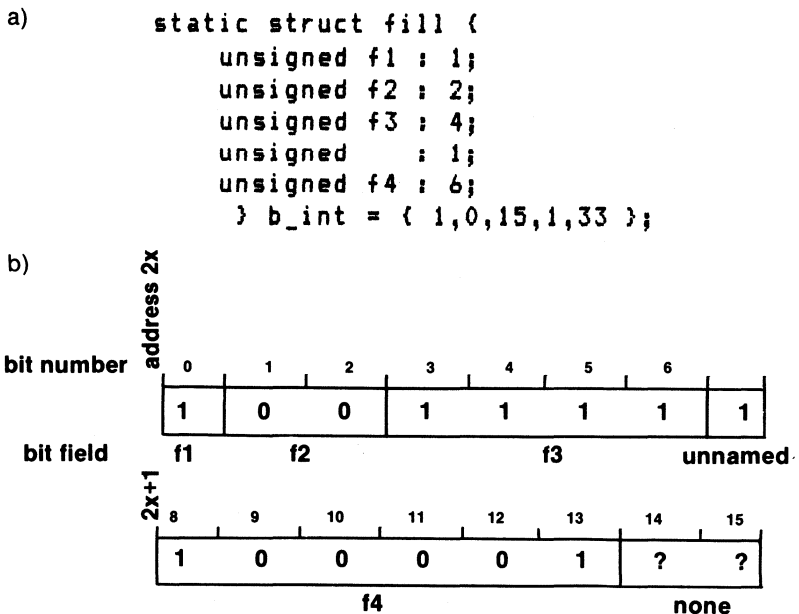
Figure 10.6. a) An example of a structure exclusively containing bit fields and b) common internal representation of it. Two byte int allocation and left to right bit field storage assumed.

- The beginning address of a structure, which is also the address of its first member, always occurs on a byte boundary. Since bit fields are associated with int's, the alignment of the beginning of the structure must observe requirements for integers.

- Since this is a static declaraton, all explicitly unini- tialized trailing bit field variables are default initial- ized to zero. (The same is true for extern structures.) Here, however, bits 14 and 15 are not guaranteed to contain zero bits becaues they are not members of the structure.

- Unnamed members can also be initialized. If mem- bers following the unnamed bit fields are to be explicitly initialized, then the unnamed bit fields must be assigned initializers to preserve the one to one matching scheme.

In addition, no field is allowed to "straddle" an integer address. For example, suppose that in figure 10.6, **b_int** was declared with the member:

```
unsigned f5 : 5;
```

added after **f4**. The new bit field would not occupy bit positions 14 and 15 of the current int and positions 0 and 1 of the next int. Instead it would occupy positions 0 through 3 of the next byte to avoid straddling the new integer field.

The Use of Bit Field Variables

Bit field variables may be used as ordinary unsigned int's with a few exceptions. The most obvious concerns the range of values that can be held by a field; fields have lower numeric bounds of stored data that reflects their length. Specifically the bounds of a field range from 0 to $n^2 - 1$, where n is the size in bits of the field. Secondly, fields do not have addresses in the ordinary sense (i.e. no integral byte address), so the address of operator, **&**, may not be applied to bit field variables.

Program 10.4, illustrated in figure 10.7, demonstrates a very simple use of bit fields and unions. The union **i_input** is composed of an integer and **f_val**, a structure of type i_parts. **f_val** divides the integer field into a sign bit and a field containing the value bits. Since the union was defined so that these two bit fields overlap exactly with the integer **val**'s corresponding parts, the value and sign fields of **val** can be directly manipulated. The sign bit of **val** is changed by lines 18 through 21. This causes the bit string comprising **val** to be interpreted as the two's complement of the original value.

```
/*      Program 10.4 - example of the use of
   structures containg bit fields and practi-
   cal use of unions. Very system dependent. */

struct i_parts  {               /* 5 */
    unsigned sign : 1;
    unsigned val  : 15;
                  };
main()
                                /* 10 */
{   static union i_compl {
        int val;
        struct i_parts  f_val;
                  } i_input;
                                /* 15 */
    printf("\nInput an int quantity: ");
    scanf(" %d ", &i_input.val);
    if (i_input.f_val.sign)
       i_input.f_val.sign = 0;
    else                        /* 20 */
       i_input.f_val.sign = 1;
    printf("\nThe complement of the input");
    printf("\n     value is:  %d",i_input.val);
}
```

program output of following page

```
Input an int quantity: 0

The complement of the input
     value is:   -32768
```

Figure 10.7. System dependent program illustrating the use of bit field variables.

11

The C Standard Library

Introduction

References to the C standard library have been made throughout this text. A library is nothing more than one or more files that contain a group of predefined functions and routines. (A routine is one or more related functions that operate together to perform an specific goal.) Since C does not contain many conventional routines such as I/O, file handling, etc. a library of useful functions has evolved with C.

443

The adjective "standard" has become somewhat misleading. In C's formative years at Bell Laboratories, the library was indeed standard because C was only available on the Unix® operating system. Unix still maintains close standards on C. However, over the years three conditions have contributed to the slow erosion of uniformity once enjoyed among C libraries:

- New routines have been added to the Unix library routines by Bell Labs. By this token, older compilers cannot and some newer compilers will not include these newer routines. This generally is not a serious problem as routines can be easily added to most libraries.
- Compilers that run on different operating systems will naturally conform to the constraints and exercise the advantages of each environment. Often this has led to the transfiguration of function call syntax or the adoption of totally different routines.
- New non-standard functions are often added to the standard library file by compiler manufacturers. While product differentiation is often a good sales practice, such routines should either be placed in a separate library or in some way be identified as non-standard.

Failure to maintain an externally defined standard has lead to the proliferation of many dialects of a language with the resulting inevitable clarity and transportability problems. The degree of standardization of the library should be an important factor in the selection of a C compiler, although by no means should it be the only factor.

Access to routines within the standard library is accomplished by different methods, depending upon the compiler and required function. Some compilers automatically search libraries for occurrences of called functions. In most, the programmer must explicitly state the library file name during the linking process. Then only the required functions will be included in the executable program. The simpler routines in C

are often implemented by a macro expansion. Pseudofunctions of this type reside, by convention, in files with the extension .H, generically called "header files". Directives that reside in header files must be inserted in the C source file through the use of **#include** directives. All unused macros in a source file are deleted in the preprocessor phase of the compiler,.

The remainder of this chapter is devoted to C standard library routines. Much of it is written in a reference style. No attempt has been made to be exhaustive; only the more important and commonly used routines have been included. The individual library routines begin with a "synopsis" section that illustrates the form that the function header takes. For example, the function header for strcpy() looks like:

char *strcpy(str1, str2)
char *str1, *str2;

From this header, both the form of the function call, which is

strcpy(_string-pointer1, string-pointer2_**)**

and the returned type, which is a pointer to type char can be determined.

Inevitably, there will be some variations between compilers in the way each actually implements "standard" library functions. The final source on library functions will be the compiler documentation, or the actual library source code if available.

I/O ROUTINES

I/O routines deal with the transfer of data to peripheral devices such as the monitor, keyboard, printer, secondary storage, etc. Many of these routines deal with the standard input or output devices, which are often defined as the terminal keyboard and monitor by default. The standard device defini-

tions can be changed by a process called *redirection*. The manner in which redirection is implemented is system dependent, although command line arguments are often utilized.

In a similar manner, it is possible to redirect the standard device definitions so as to read and write to specific files. Many files use a special marker, called an *end-of-file* (EOF) value, that designates the end of the file. Unfortunately some environments use 0 as the EOF value while others use -1. Many of the following I/O functions will return an EOF value upon encountering either an error condition or the end of the file.

Character Input and Output

Because C is foremost a systems language, the most fundamental I/O routines deal with individual character values. To access these functions, it is necessary to include the standard I/O library header file through the directive:

#include <STDIO.H>

STDIO.H is a header file that contains the macro expansions commonly used in I/O routines and file manipulations.* The two character I/O (pseudo) functions getchar() and putchar() are normally implemented as macros.

int getchar()

Accepts (reads) an individual character value from the standard input stream and returns the encoded data as an int (high byte(s) are set to zero). This function returns an EOF if an error occurs. An EOF value is also returned when the end of the file is reached during file input redirection.

---

* On Unix and some other operating systems, filenames are designated in lowercase. As a result the directive would be #include <stdio.h>.

int putchar(c)
char c;

Sends (writes) the character **c** to the standard output device. putchar() returns the value **c** if successful, otherwise if an error is encountered, it returns EOF.

Unformatted String I/O

On the next higher level, a plain string of characters can be sent or received from the standard I/O device. The actual argument of these string output functions must be a character pointer value. An actual occurrence of a string and a char array name also resolve to a char pointer value. The first occurrence of a null value normally terminates a string.

char * gets(str_buf)
char *str_buf;

This function reads characters from the standard input stream until a newline (or return) is read. The newline character is suppressed, a null is appended to the end, and this string is stored in memory in the *buffer* pointed to by the argument. A buffer is just a block of memory set aside for the express purpose of storing I/O information. Typically the actual argument is a pointer to the beginning of character array of appropriate size, declared in the calling function. It is the responsibility of the programmer to be certain that the buffer is at least as large as the input string. gets() returns the pointer value to this string if successful, otherwise an EOF value is returned if an error condition occurs or no characters are read (i.e. a null string).

```
int puts(str_ptr)
char *str_ptr;
```

Writes the string (i.e. the characters up to the first null value) pointed to by the argument to the standard output. The terminating null character is not written; however, a newline character is automatically appended to the end of the string. Upon encountering an error condition, EOF is returned.

Formatted String I/O

The two corresponding routines for formatted string output are printf() and scanf(). Function calls to these routines have been used throughout this text. Both functions are slower than the previous classes of functions because of their greater complexity. printf() and scanf() should be reserved for strings in which data type conversion is desired.

```
int scanf(format, arg1, arg2...)
char *format;
union PTR_FUND arg1, arg2,...;
```

scanf() reads characters from the standard input and interprets these characters according to the conversion string pointed to by **format**. These interpreted values are assigned to the actual arguments through indirection.

The actual arguments must be pointers to an (extended) fundamental type variable, while the matching formal arguments **arg1, arg2**, etc, are union variables of type PTR_FUND. This data type is externally type defined in the file STD10.H. scanf() returns the number of arguments successfully converted, otherwise EOF upon an error condition or end of file.

The scanf() function call syntax is covered in chapter 5.

```
int printf(format, arg1, arg2,...)
char *format;
union PTR_FUND  arg1, arg2,...;
```

printf() is the complement to the scanf() function. This function writes characters to the standard output according to the conversion string referenced by pointer **format**. This function was covered in chapter 3. In review, the following converson specifications can appear in the corresponding function call:

%**d** decimal (base 10)
%**x** hexadecimal (base 16)
%**c** character
%**e** floating point decimal in exponential format
%**f** floating point decimal in plain format
%**g** %e or %f, whichever is shorter
%**o** octal (base 8)
%**u** unsigned decimal
%**s** string

The following modifications may also be used between the % and specification character:

- **-** Specifies that the numeric value will be left justified.
- **+** Specifies that the numeric value will be output with a indicative positive or negative sign.
- **#** Causes octal integer values to be preceded by a leading zero and hexadecimal values to be preceded by a leading zero and x.
- *ff.pp* *ff* is a decimal number that specifies the field width. The field width is the number of character positions used for output. Padding with spaces will occur to meet width requirements. If a * occurs in this field, then the next argument specifies the width field. *pp* specifies the precision. This quantity is interpreted as the maximum number of positions for a string, the minimum number for an integer, and the number of digits to the right of the outputted decimal point of a floating point value.
- **l** Specifies the long extension for the d, o, x, and u formats.

If the character following the % character in the conversion string is not a valid conversion character, it is output un-

changed. Hence %% outputs a single percent character.

Later versions of this function return the number of characters successfully written to output, or an EOF upon error.

File I/O

There are two methods of accessing a file in secondary storage for I/O operations. One method has been mentioned; it involves redirection of a program's I/O to a file. The other method is more flexible; it involves the use of special versions of the functions just presented: fgetc() (corresponds to getchar()), fputc() (corresponds to putchar()), fgets(), fputs(), fscanf(), and fprintf(). Each of these functions include one more argument than its counterpart. This argument is a pointer to the object file.

In fscanf() and fprintf() the pointer argument appears first. fscanf()'s header would appear as:

```
int fscanf(file_ptr, format, arg1, arg2, ...);
FILE *file_ptr;
char *format;
union PTR_FUND  arg1, arg2, ... ;
```

where the FILE "data type" is actually an externally typedefined structure template found in STDIO.H. In the other four file I/O functions, the file pointer is the last (or only) argument. Pointer values to normal files are obtained when the indicated file is opened for access, as will be shown in the section entitled "File Handling Routines".

There are two specific pointer constants of type FILE that are macro defined in STDIO.H — stdin and stdout. These simple macros point to the standard input and output devices. They can, however, be used wherever a file pointer can. Consequently the following function call is equivalent to its less complicated printf() counterpart:

```
fprintf(stdout, "Hello %d you", 2);
```

Aside from the extra file pointer argument which specifies output to a specific file, these functions behave exactly like their non-file counterparts. The one exception is that fputs() does not append a newline character to the output string as does puts().

IN-MEMORY FORMAT CONVERSION ROUTINES

Two routines are provided for performing the in-memory conversion between the binary representation of a fundamental type data item and corresponding character string representation. These two routines are sscanf() and sprintf(). They are closely related to the fscanf() and fprintf() routines, except that the former two routines do not perform any sort of I/O, but rather perform their respective conversion routines among two or more objects in main memory. Their first argument is not a pointer to a file but instead a pointer to a previously declared string or character array.

```
int sscanf(str_ptr, format, arg1, arg2)
char *str_ptr;
char *format;
union PTR_FUND arg1, arg2,...;
```

The characters indicated by the conversion specifications in **format** are "read" from the string referenced by **str_ptr**. These characters are converted to the form appropriate to the corresponding conversion specification. Then these converted values are ordinally assigned to the actual arguments of the sscanf() function call through the process of indirection. Aside from the first actual argument, the syntax of the corresponding function call mimics that for scanf(). For example, the call:

```
sscanf(line, "arg[1] = %d", val);
```

would expect the first non-white characters in string **line** to be
ary[1] = followed by a group of characters representing a
decimal integer value. If this indeed is the case, the numeric
character group is converted to an int quantity and assigned to
val.

The number of items successfully assigned is returned by
this function.

> **int sprintf(str_ptr format, arg1, arg2...)**
> **char *str_ptr;**
> **char *format;**
> **union PTR_FUND arg1, arg2,...;**

This function is the complement to sscanf(). sprintf()
assigns the string specified by the **format** conversion string to
the block of memory pointed to by **str_ptr**. The associated
function call uses many of the same syntax conventions as a
call to printf(). As an example, if the value of the integral
variable **rad** was ten, the following call would place the string
"The correct radix is 10" at the block of memory pointed to be
str1.

```
sprintf(str1, "The correct radix is %d", rad);
```

A null character is automatically appended to the end of the
assigned string. Again it is up to the programmer to insure that
the accepting memory block is large enough to hold the
assigned string, including the added null. printf() returns the
number of characters successfully "written", or a zero if an
error condition is encountered.

CHARACTER ROUTINES

The standard library functions available for character
manipulation are very elementary, all being either class tests or
alphabetic case transformations. Often these functions are

implemented as argumented macros in a header file conventionally named CTYPE.H.

Class Tests

This group of functions accepts a single char argument and uses comparison operations to determine if this character is associated with a particular class. If the character is of the specified class, a value of int true (non-zero, usually one) is returned. A returned value of zero indicates the character is not of that class and the character is described as having failed the class test. When implemented as true functions, the function headers of the routines have the form:

```
int function-name(c)
char c;
```

Function Name	Class Description/ASCII Definition
isalnum	alphanumeric /48 through 57, 65 through 90, or 97 through 122
isalpha	alphabetic /65 through 90, or 97 through 122
isascii	standard ASCII / 0 through 127
iscntrl	control character / 0 through 31, or 127
isdigit	numeric digit / 48 through 57
isgraph	graphics character / 33 through 126
islower	lowercase alphabetic / 97 through 122
isprint	standard printing character (includes space) 32 through 127
ispunct	punctuation / 32 through 47, or 58 through 64
isspace	whitespace /32, 9, 10, or 13 (sometimes include form feed 12)
isupper	uppercase alphabetic / 65 through 90
isxdigit	hexadecimal digit / 48 through 57, 65 through 70, or 97 through 102

Alphabetic Case Transformations

The functions tolower() and toupper() perform the processes of "de-capitalization" and capitalization, respectively. They are also often implemented as macros.

If the **tolower(c)** function's argument is an uppercase alphabetic character, this function returns the lowercase equivalent. Otherwise it returns the same value.

If the **toupper(c)** function's argument is a lowercase alphabetic character, this function returns the uppercase equivalent. Otherwise it returns the same value.

STRING ROUTINES

The ability to manipulate character strings efficiently is an important attribute for a systems language. String-handling is not only paramount in text related programs such as editors and word libraries, but this ability is central to compilers, assemblers, and other related programs. For example, a source program is nothing more than a long sequence of somewhat arbitrary characters. A compiler's first job is to separate this sequence into simpler strings for further processing. C is a good systems language partly because of the simplicity and efficiency of its string "operations".

The following routines perform operations on null-terminated character strings through a pointer to a beginning of such a string. A pointer value to a character string can take the form of a character pointer, a char array name, or the actual occurrence of a double quote enclosed string. The function argument declarations have been left out of the following headers to avoid repetition. The arguments have the following consistent declarations:

```
char *str_ptr1, *str_ptr2;
char *str_ptr;
char c;
int n;
```

char *strcat(str_ptr1, str_ptr2)

Concatenates the string pointed to by the second argument onto the end of the string pointed to by the first. The first string should have a declared length sufficient to hold the concantenation of the two. This function returns the value of **str_ptr1**.

char *strchr(str_ptr, c)

Searches the indicated string for the *first* occurrence of the character **c**. If such an occurrence is found, the function returns the pointer value to it. The null pointer is returned if **c** does not appear in this string.

int strcmp(str_ptr1, str_ptr2)

Compares the two indicated strings, character by character until an ordinally corresponding pair of characters are not identical. The string with the lower valued first non-identical character is said to be less than the other. Two strings are equal only if they have identical characters in the same order and are of the same length. This function returns:

-1 if string 1 < string 2
 0 if strings are equal
 1 if string 1 > string 2

char *strcpy(str_ptr1, str_ptr2)

Copies the second indicated string "over" the first indicated string. String 1 should be at least as long as string 2. If the first is longer, it will retain some of its trailing characters. (The null terminator from the second string is not copied.) Returns pointer value **str_ptr1**.

int strlen(str_ptr)

Returns the length, in number of characters, of the indicated string. The null terminator character is not interpreted as a string element.

char *strncat(str_ptr1, str_ptr2, n)

Operates in the same manner as strcat() when the second string has **n** or fewer characters. If the second string has more, only the first **n** characters will be concatenated to the end of the first string.

int *strncmp(str_ptr1, str_ptr2, n)

Operates in the same manner as strcmp() when string 2 has **n** or fewer characters. If it has more, then only the first **n** characters are used in the string comparison.

char *strncpy(str_ptr1, str_ptr2, n)

Operates in the same manner as strcpy() except that only up to **n** characters are copied over string 1.

char *strrchr(str_ptr1, c)

strrchar() is the complement to strchr() in that it searches for the *last* occurrence of character **c** in the indicated string. It returns a pointer to this last occurrence. If this character is not encountered, it returns the null pointer. strrchr() and strchr() will return the same value when there is either zero or one occurrence of the search character in the string.

In quite a few library implementations, the string routines strcat(), strcpy(), strncat(), and strncpy() do not return the pointer value to the first string. This value is actually the first argument (**str_ptr1**); consequently it is already known to

the calling function. This returned value is just a convenience in that it allows the function call to be directly used in an operational statement.

FILE HANDLING ROUTINES

This section details some of the standard library routines for manipulating files other than that currently executing. It is frequently useful for a program, when executing, to open another file, read data from it, manipulate this data, write data back to this file, and then close it. Most computer languages have the ability to perform basic file operations of this sort.

C's file handling routines actually form a subset of the I/O routines, since accessible files are located in secondary storage (generally disk). File handling routines often utilize the capabilities of the operating system. Because C was developed in conjunction with Unix, many of C's library file routines reflect the file capabilities of Unix. The same routines under other operating systems must be accomplished using somewhat different methods, however these differences should for the most part be invisible to the user.

Buffers and High Level Routines

A buffer is a block of memory in which data is temporarily stored during transfer of this information between physical locations. Many C library routines, including those discussed in this section, utilize blocks of main memory, in the form of an array or structure, as buffers. Because the use of buffers allows a smoother and more controlled exchange of data between functions, the C I/O routines that make use of buffers are often referred to as "buffered or high-level I/O".

This title also reflects the fact that these routines are "built upon" low-level functions. Low-level functions represent a more direct access to the operating system. Because they are not buffered, low-level I/O functions handle data in either

single units or in blocks whose size reflects the convention for the operating system or the physical block size of secondary memory.

Buffers also increase the efficiency of data transfer, especially between main memory and any peripheral device. This is due to the fact that overhead time is associated with each use of an I/O device. Buffers allow small groups of data to be stored and collectively transferred. Consequently the number of I/O calls and the associated total overhead is reduced. The process of emptying a buffer, usually to a file in secondary storage, is called "flushing a buffer."

File Parameters

A file in secondary memory has at least three system parameters associated with it:

- A filename, which can be divided into a primary name and an optional period and filename extension. For example, the filename PROG2.C has a primary name of PROG and an extension of C.
- A location on the secondary storage medium.
- A total length.

Of course these parameters can be changed through the disk operating system (DOS), but at any one time these parameters are constant. When the operating system needs to reference a file, it associates an integer number, called a *file descriptor*, with each accessed file. The operating system uses this number instead of the filename to identify the file.

When a C routine handles files, the file description along with other information about the file — the mode in which it was opened (see fopen()), the character position within the file of the current operation, the location of the associated buffer, the number of characters in the buffer, etc. — are stored for each file in a structure flavored variable declared through the

library routine. These variables are of type extern FILE where this declarator is a typedef defined in STDIO.H.

As exemplified by the fprintf() and the fscanf() functions, the use of many file handling functions requires an argument of the type "pointer to type FILE" (e.g. **file_ptr**). Actually such an argument does not point to a file, but rather to a structure variable (of type FILE) that contains relevant data on the file. The thing that is important from the user's viewpoint is that this value is required for most file manipulations, and it is supplied as a returned value when a file is opened.

Opening and Closing a File

Before the data in a different file can be accessed by the executing program, the file must be opened. After all operations on a file have been completed, the file should be formally closed before the termination of the program. Closing a file insures that all outstanding information associated with the file is flushed from the buffers and proper system housekeeping is maintained. There are four functions associated with the normal opening and closing of files.

FILE *fopen(filename, mode)
char *filename, *mode;

fopen() opens the file specified by pointer **filename** according to the manner specified by the second argument — the access mode. The supported access modes are:

r	The existing file is opened to the beginning for reading of characters.
r+	The existing file is opened to the beginning for both reading and writing.
w	If the file exists it is opened to the beginning for writing; if it does not exist it is created for write access.

Overwriting of old information is permitted.

w+   Same as **w** except both reading and writing are permitted.

a   If the file exists it is opened to the end for appending information. If it does not exist it is created for writing. Overwriting of old information is not permitted.

a+   Same as **a** except reading and writing are permitted.

fopen() returns a pointer to (the structure pertaining to) the opened file. If the call is unsuccessful, the null pointer is returned. For example, the first statement:

```
FILE *fopen(), *fp1, *fp2;
fp1 = fopen("PROGZ.C", "a+");
```

declares **fp1** and **fp2** to be pointers to a FILE structure variable, and **fopen()** to be a function returning a pointer to type FILE. The second opens file PROGZ.C for read/append access and assigns **fpr1** the pointer value associated with this opened file. It is possible to have more than one file open at any one time but each corresponding FILE pointer value must be preserved until its file is closed.

int fclose(file_ptr)
FILE *file_ptr;

fclose() flushes the buffers associated with the file specified by **file_ptr**, then closes the file. 0 is returned if the procedure was successful; a value of -1 is returned upon encountering an error. For example, given the file opening statements presented under fopen(), the statement:

```
fclose(fp1);
```

closes the file PROG.Z and returns a value of 0, which is discarded. Any flushed file buffers are returned to the heap for possible reuse.

FILE *tmpfile()

Creates and opens a *temporary* file in "w+" mode. Temporary files are automatically deleted when a program terminates in a normal fashion. These files usually serve a purpose somewhat related to buffers in that they are temporary areas for file data storage. However, data in buffers cannot be easily accessed and manipulated, while the data in a temporary file can be manipulated in the same manner as permanent files. tmpfile() returns a pointer to type FILE on success; otherwise it returns the null pointer. Temporary files generally are not given names; however some libraries supply the function tmpnam() for just such a purpose.

freopen(filename, mode, file_ptr)
char *filename, *mode;
FILE *file_ptr;

This function combines the effects of a fclose() and fopen() in a very useful way. If successful, the file associated with **file_ptr** is closed and the file associated with **filename** is opened according to the mode specified in the second argument (see fopen()). In addition, the newly opened file is associated with the pointer to the closed file; the FILE pointer value is reused. Often a call of this sort is used to implement redirection. As an example, the statement:

```
fp2 = freopen("OUT.DAT", "w+", stdout);
```

redirects the standard output to file OUT.DAT. Statements of this sort allow a finer control of redirection than command line arguments, since they allow redirection to occur repeatedly throughout the program. freopen() returns the pointer associated with the newly opened file (the third argument) upon success; the null pointer is returned if an error occurs.

Because both the **stdin** and **stdout** device pointers can be redirected through this function, libraries provide the device "filename" **stderr** for direct, unqualified access to the monitor. **stderr** should never be redirected. **stderr** therefore is often used for the destination of error messages.

File I/O

Reading out of and writing to a file is accomplished chiefly through calls that are analogous to their device I/O counterparts. These routines are described under the earlier section entitled "I/O Routines". These six file I/O routines and their analogous device I/O routines are listed in table 11.1. The new routines fflush(), fread() and fwrite() are described afterward.

Table 11.1. Listing of commonly used file I/O and corresponding device I/O routines.

File I/O Routine	Corresponding Device Routine
int fgetc(file_ptr)	getchar()
int fputc(c, file_ptr)	putchar()
char *fgets(str_buf, file ptr)	gets()
int fputs(str_buf, file_ptr)	puts()
int fscanf(file_ptr, format, arg1,...)	scanf()
int fprintf(file_ptr, format, arg1,...)	printf()

int fflush(file_ptr)
FILE *file_ptr

This function "flushes" or outputs any data outstanding from internal buffers associated with the FILE pointer to the file associated with this same pointer. In simpler terms, fflush() clears the buffers and updates the indicated file with this information. A call of this sort is often performed if some lengthy or complex code is executed between file writing calls. In this manner, if the intervening process causes an abnormal program termination, then the information in a flushed file will have already been written to file and will probably not be lost as would happen if it remained in the buffer. This function returns zero if it executes successfully; minus one is returned if an error is encountered. The flush process automatically occurs during every invocation of a device I/O routine.

fread(str_buf, size, n, file_ptr)
char *str_buf;
unsigned size, n;
FILE file_ptr;

Copies **n** data items, each of length **size**, from the indicated file into a buffer set up by the executing program. The routine returns the number of items read, which if totally successful will equal **n**. The most efficient length for this routine is machine dependent.

fwrite(str_buf, size, n, file_ptr)
see fread

Complement to fread(). Writes **n** data items of individual length **size** from a program buffer to the indicated file. Returns the number of items successfully written.

Location Within a File

Files often have some type of logical format imposed on them. For example, perhaps information was entered into a file so that certain columns always contain specific numeric values. This occurs when storing a series of records (structures).

The standard library has a number of functions concerned with locating the current read/write position in terms of an *offset*. An offset is the number of bytes between the current character and a reference position.

long ftell(file_ptr)
FILE *file_ptr;

ftell() returns the offset, in bytes, of the current read/write position from the beginning of the file. On error, a -1 is returned. Opening a file in the read or write modes causes the current position to be zero offset, while an append offset is the old length of the file.

int feof(file_ptr)
FILE *file_ptr;

feof() returns a non-zero value if the identified file has reached the end-of-file marker, otherwise zero is returned. Under some systems, files opened in update modes (i.e. modes where writing is possible), only have an EOF appended when the file is closed.

long fseek(file_ptr, offset, mode)
FILE *file_ptr;
long offset;
int mode;

fseek() relocates the current read/write position of the indicated file to a position located **offset** bytes away from a

reference point specified by **mode**. The allowable values for **mode** are:

0   offset is relative to the beginning of the file
1   offset is relative to the current position within the file
2   offset is relative to the end of the file

If the requested repositioning is successful, a zero is returned; otherwise a non-zero value is returned. It is illegal to request a position beyond the current bounds of a file.

void rewind(file_ptr)
FILE *file_ptr;

rewind() relocates the current read/write position of the indicated file to the beginning of that file.

Error Handling

Operating systems associate an error code status with each file. A status of zero indicates no error has been associated with the file, while a non-zero value indicates an error has been detected. An error can occur during any file routine, due to any number of causes:

- Trying to open with an invalid filename
- Referencing an unopened file
- Illegal argument type or value
- Exceeding a file's legal bounds
- Improper file format

int ferror(file_ptr)
FILE *file_ptr;

ferror() reports the error status associated with the indicated file. It returns a non-zero if an error has been detected up

to that point during the current program execution. Otherwise a zero, indicating a past error, is returned.

int clearerr(file_ptr)
FILE *file_ptr;

Resets the error status associated with the indicated file to zero. This indicates that no past error is associated with this file. Note that this routine does not correct conditions that caused any errors; it only resets a flag. Returns a non-zero if successful, otherwise a zero.

void exit(end_val)
int end_val;

This routine is used to terminate execution of a program. It performs two basic services. First, it executes a call to fclose() for each open file. Consequently all outstanding data is flushed from the buffers into the associated files, and the files are properly closed. Secondly, this function in turn calls a function named _exit(), which actually performs the regular termination procedure. The value **end_val** is presented to the invoking program (often the operating system). By convention, non-zero end values denote abnormal program termination.

DYNAMIC MEMORY ALLOCATION FUNCTIONS

Dynamic allocation is the ability of a language to calculate and assign, during execution, the memory space required by objects in a program. C does not inherently have this facility. For example, array subscript declarators must be constant expressions. The following functions deal with allocating and freeing regions of dynamic memory. If during allocation, a block of the requested size cannot be found, a "no core left" error condition results.

char *calloc(n, size)
unsigned n, size;

If successful, calloc() allocates a contiguous region in the heap with a length of **n * size** bytes. This region is initialized to zeroes. The function returns a pointer to (the beginning of) this block. If unsuccessful, it returns the null pointer.

char *malloc(size)
unsigned size;

malloc() allocates an unitialized, contiguous region in the heap of length **size** bytes. malloc() returns the pointer to this region if successful; otherwise the null pointer is returned.

char *realloc(dm_ptr, size)
char *dm_ptr;
unsigned size;

realloc() changes the length of a previously dynamically allocated block to the new length **size**. It preserves the contents of the retained portion even though this routine often involves physically moving the entire logical field to a new physical address. realloc() returns the address of the new region if successful; otherwise a zero is returned if a minimum sized block could not be found.

void *free(dm_ptr)
char *dm_ptr;

free() frees the block of main memory pointed to by the argument. To be released, this area must have been dynamically allocated by calloc(), malloc(), or realloc().

Because dynamically allocated blocks of memory are not given proper variable names by the compiler, information in these regions must be handled via pointers.

Appendix A.
Base Conversion Table

The following table contains the numeric interpretations of selected positive integer values. The four bases presented are extensively utilized in computer science. The fifth column illustrates the binary coded decimal (BCD) representation of a value. BCD is not a true base, but is a subset of base two that is more commonly associated with languages other than C.

Binary (base 2)	Octal (base 8)	Decimal (base 10)	Hexadecimal (base 16)	BCD
0000	000	0	0	0000
0001	001	1	1	0001
0010	002	2	2	0010
0011	003	3	3	0011
0100	004	4	4	0100
0101	005	5	5	0101
0110	006	6	6	0110
0111	007	7	7	0111
1000	010	8	8	1000
1001	011	9	9	1001
1010	012	10	A	0001 0000
1011	013	11	B	0001 0001
1100	014	12	C	0001 0010
1101	015	13	D	0001 0011
1110	016	14	E	0001 0100
1111	017	15	F	0001 0101
0001 0000	020	16	10	0001 0110
0001 0001	021	17	11	0001 0111
0001 0010	022	18	12	0001 1000

Binary (base 2)	Octal (base 8)	Decimal (base 10)	Hexadecimal (base 16)	BCD
0001 0011	023	19	13	0001 1001
0001 0100	024	20	14	0010 0000
0001 0101	025	21	15	0010 0001
0001 0110	026	22	16	0010 0010
0001 0111	027	23	17	0010 0011
0001 1000	030	24	18	0010 0100
0001 1001	031	25	19	0010 0101
0001 1010	032	26	1A	0010 0110
0001 1011	033	27	1B	0010 0111
0001 1100	034	28	1C	0010 1000
0001 1101	035	29	1D	0010 1001
0001 1110	036	30	1E	0011 0000
0001 1111	037	31	1F	0011 0001
0010 0000	040	32	20	0011 0010
0010 0001	041	33	21	0011 0011
0010 0010	042	34	22	0011 0100
0010 0011	043	35	23	0011 0101
0010 0100	044	36	24	0011 0110
0010 0101	045	37	25	0011 0111
0010 0110	046	38	26	0011 1000
0010 0111	047	39	27	0011 1001
0010 1000	050	40	28	0100 0000
0010 1001	051	41	29	0100 0001
0010 1010	052	42	2A	0100 0010
0011 0000	060	48	30	0100 1000
0011 1000	070	56	38	0101 0110
0100 0000	100	64	40	0110 0100
0101 0000	120	80	50	1000 0000
0111 0000	140	112	70	0001 0001 0010
1010 0000	200	160	A0	0001 0110 0000
1 0000 0000	400	256	100	0010 0101 0110

Appendix B.
ASCII Character Codes

The following table lists the American Standard for Coded Information interchange (ASCII) character codes in decimal and octal and the associated character symbol or control character. Character codes 0 through 31, and 127 decimal are control characters.

Decimal	Octal	ASCII		Decimal	Octal	ASCII
0	000	NUL		29	035	GS
1	001	SOH		30	036	RS
2	002	STX		31	037	US
3	003	ETX		32	040	Space
4	004	EOT		33	041	!
5	005	ENQ		34	042	'
6	006	ACK		35	043	#
7	007	BEL		36	044	$
8	010	BS		37	045	%
9	011	HT		38	046	&
10	012	LF		39	047	'
11	013	VT		40	050	(
12	014	FF		41	051	)
13	015	CR		42	052	*
14	016	SO		43	053	+
15	017	SI		44	054	,
16	020	DLE		45	055	-
17	021	DC1		46	056	.
18	022	DC2		47	057	/
19	023	DC3		48	060	0
20	024	DC4		49	061	1
21	025	NAK		50	062	2
22	026	SYN		51	063	3
23	027	ETB		52	064	4
24	030	CAN		53	065	5
25	031	EM		54	066	6
26	032	SUB		55	067	7
27	033	Escape		56	070	8
28	034	FS		57	071	9

Decimal	Octal	ASCII
58	072	:
59	073	;
60	074	<
61	075	=
62	076	>
63	077	?
64	100	@
65	101	A
66	102	B
67	103	C
68	104	D
69	105	E
70	106	F
71	107	G
72	110	H
73	111	I
74	112	J
75	113	K
76	114	L
77	115	M
78	116	N
79	117	O
80	120	P
81	121	Q
82	122	R
83	123	S
84	124	T
85	125	U
86	126	V
87	127	W
88	130	X
89	131	Y
90	132	Z
91	133	[
92	134	\

Decimal	Octal	ASCII
93	135	]
94	136	^
95	137	_
96	140	
97	141	a
98	142	b
99	143	c
100	144	d
101	145	e
102	146	f
103	147	g
104	150	h
105	151	i
106	152	j
107	153	k
108	154	l
109	155	m
110	156	n
111	157	o
112	160	p
113	161	q
114	162	r
115	163	s
116	164	t
117	165	u
118	166	v
119	167	w
120	170	x
121	171	y
122	172	z
123	173	{
124	174		
125	175	}
126	176	~
127	177	del, rubout

Appendix C.
The Binary Number System and the
Complement Form of Negative Numbers

The numbering system most commonly used is the decimal number system, which is based on ten digits, zero through nine. However, since digital circuitry is bistable (i.e. capable of attaining only two states), digital logic requires the use of the base two or the binary number system.

The binary number system is based on two digits, zero and nine. The easiest way to understand how a value can be represented in binary is to contrast the binary and decimal representation of a value, and to illustrate what each representation actually symbolizes. For this purpose, consider a value of 215.75_{10}, where the subscript denotes the base. This value can be represented in decimal as:

$$2 \times 10^2 = \quad 2 \times 100_{10} = 200$$
$$1 \times 10^1 = \quad 1 \times 10_{10} = \quad 10$$
$$5 \times 10^0 = \quad 5 \times 1_{10} \quad = \quad 5$$
$$7 \times 10^{-1} = \quad 7/10_{10} \quad = \quad 0.7$$
$$5 \times 10^{-2} = \quad 5/100_{10} \quad = \quad 0.05$$

adding components gives   215.75

The binary representation of this number is 11010111.11. It could also be broken up into pairs of ten. However these powers should be in base two. (For example, 10^2 in binary would be 2 x 2 or 4 in decimal.) Since most people are not trained to think in terms of binary, the decimal interpretation of a binary number is much more useful:

Binary Interpretation		Decimal Conversion		
$1 \times 10^7 =$	10000000_2	$= 1 \times 2^7 =$	128_{10}
$1 \times 10^6 =$	1000000_2	$= 1 \times 2^6 =$	64_{10}
$0 \times 10^5 =$	0_2	$= 0 \times 2^5 =$	0_{10}
$1 \times 10^4 =$	10000_2	$= 1 \times 2^4 =$	16_{10}
$0 \times 10^3 =$	0_2	$= 0 \times 2^3 =$	0_{10}
$1 \times 10^2 =$	100_2	$= 1 \times 2^2 =$	4_{10}
$1 \times 10^1 =$	10_2	$= 1 \times 2^1 =$	2_{10}
$1 \times 10^0 =$	1_2	$= 1 \times 2^0 =$	1_{10}
$1 \times 10^{-1} =$	0.1_2	$= 1 \times 2^{-1} =$	0.5_{10}
$1 \times 10^{-2} =$	0.01_2	$= 1 \times 2^{-2} =$	0.25_{10}
	11010111.11_2	$=$	215.75_{10}

Binary arithmetic is much more simple than decimal arithmetic since there are only two binary digits. The three rules for digit-wise addition for two binary numbers can be stated mathematically as:

$$0_2 + 0_2 = 0_2$$
$$1_2 + 0_2 = 1_2$$
$$1_2 + 1_2 = 10_2$$

where the normal associative and commutative laws apply. For example, the addition of 10101_2 and 11_2 yields 11000_2.

Negative numbers in any base can be represented by a form called the complement (in that base).

If x is some number with n significant digits, and is of base b then the complement of x in that base can be determined by the equation:

$$(x - b^n) = b's\ complement$$

As an example, the ten's complement (i.e. complement in base ten) of 37_{10} is $37-10^2 \Rightarrow 63_{10}$. If one takes into account the b^n term, it is possible to use the complementary form in place of a

negative number. 100+(-37) could be written as 100+63-100.

A shortcut method for determining the two's complement of a binary number requires two steps:

1. Working off the original binary number, replace all the ones with zeroes, and zeroes with ones.
2. Add one to the result of step 1.

For example, the complement of 110100_2 would be obtained by the steps:

binary number	0110100_2
1. switch numerals	1001011
2. add one	1
two's complement	1001100

Note that the leading zero and one denote the standard and complement notation, respectively. The result of the first step, here 1001011, is known as the one's complement of 110100. (This is a misnomer, as the name does not imply base one.)

Integer values are stored in memory as binary values, with one digit per bit. The leftmost bit of a data block can be interpreted in one of two ways:

- As just another digit. This interpretation corresponds to an unsigned integral data type.
- As an indicator of complementation (i.e. a sign bit). This corresponds to a signed integral data type.

Assuming a one byte int length for simplicity, the storage block:

0	1	2	3	4	5	6	7	bit number
1	0	0	1	0	1	1	1	

could be interpretted as either the unsigned quantity 151_{10} or the signed quantity -105_{10}.

Appendix D.
Summary of C Operators

Below is a replication of table 4.1, which lists the operators of C, in order of precedence.

Class Name	Operators	Associativity*
primary	() [] ->	→
unary	! ++ -- (cast) * & sizeof −	←
multiplicative	* / %	→
additive	+ -	→
shift	<< >>	→
relational	< > <= >=	→
equality	== !=	→
bitwise and	&	→
bitwise xor	^	→
bitwise or			→
logical and	&&	→
logical or	||	→
conditional	?:	←
assignment	= += -= etc	←
sequence	,	→

(Left column margin: highest → ... Precedence ... ← lowest)
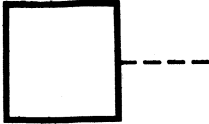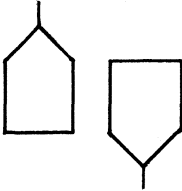
* → symbolizes left to right associativity while ← symbolizes right to left. (The binary operators *, +, &, |, and ^ have undefined associativity.)

Appendix E.
Flowchart Symbols

A flowchart is a diagram, or a type of blueprint of the logic flow of a program. Typically, it is drawn before a program is coded, as an aid to algorithm and program development. The more commonly used flowchart symbols are presented below.

Symbol	Name	Use
	Flowline	Indicates direction of control flow within a program.
	Process	A rectangle signifies one or a small number of related program statements. A description of the process usually appears in the rectangle.
	Decision	A diamond indicates conditional execution. The test condition usually appears within the diamond.
	Collector Node	This shape is used to join two flowlines into a resultant single line. Occurrences of nodes correspond to points in a program associated with non-sequential execution.
	Input/Output	A tilted parallelogram denotes an input or output operation or routine. A description of the I/O process and the physical device specification appear within.

Symbol	Name	Use
	Comment	A dotted line indicates an explanation or description of the indicated point of the flowchart. An actual description resides within square.
	Predefined Process	A capped rectangle indicates that a predefined process (such as a subroutine, function, another program, operation system action, etc,) occurs. In C, this shape usually denotes a function call in the program.
	Communication Link	Denotes the establishment and use of a telecommunications link for relaying information. Arrowheads indicate direction of data flow. (The example is a bidirectional link; communication can occur from either direction.)
	Offpage Connector	This symbol denotes that the flowchart is continued on or from another page, respectively.
	Terminal	An oval indicates the beginning or end of a program or a system interruption of the program.

Appendix F.
Derived Data Types
and Their Declaration

One of C's strong points is the fact that C allows the programmer to define his or her own derived data types. The data types supported by C are:

- fundamental data types — char, int, float, double, and enum. The int type can be optionally extended with the keywords: short, unsigned, and long.
- pointers
- arrays
- structures
- unions

The last four types are called derived types, because they are developed from the fundamental types. These categories of variables, with the addition of functions, comprise the objects of the C language. (An object is a region of storage that can be referenced and manipulated from within a program.)

In the most basic usage of derived data type variables, the members or elements are all of fundamental type (e.g. a structure containing a char, and int, and a float). However, it is possible to construct complex variable types by "nesting" derived data type definitions. The following kinds of variables and functions are allowed in C:

- A pointer to a function, array, structure, union, pointer, or a fundamental data type variable.
- An array of either structures, unions, arrays, pointers, or fundamental type variables.
- A structure (or union) containing structures, unions, arrays, pointers, fundamental type variables, and/or bit fields.

- a function returning a pointer or fundamental
  data type value. A Unix Version 7 extension
  allows passing of structure or union values.

For instance, the following statement:

```
int (*apai[10])[];
```

declares **apai[]** to be an array of ten pointers to arrays containing integers. Despite the complex nature of the declaration, **apai[]** is just an array variable. The programmer cannot actually create a radically different data type.

The basic objects in C are declared in sowewhat different manners. Pointers, arrays, and functions are declared using the associated operators — the indirection *, subscript [], and argument () operators, respectively. When one of these operators appears in a declaration, it acts only as a declarator and does not imply the actual operation. Previously defined templates are generally used to declare structures, unions, and enumerated data type variables. Functions are defined by their actual occurrence, although redeclaration of functions returning non-integer values is often necessary.

To declare a more "complex" derived variable, a combination of basic declaration techniques must be followed. Complex declarations must be coded in such a manner that the natural precedence and associativity of any operators is observed. Often associative parentheses are used in declarations to force the association of operators or templates with variable identifiers in the desired manner.

As an example, consider the following statement:

```
char *apc[9], (*pac)[], *fpc(), (*pfc)();
```

It declares: **apc** as an array of nine pointers to characters, **pac** as a pointer variable to an array of characters, **fpc** as a function returning a pointer to a character, and **pfc** as a pointer to a function returning a character value.

Complex declarations can occur at the same point as template definitions. The statement:

```
struct segmt {
    int i1, i2;
    long *lp;
    } thing1, (*(*papas)[])[];
```

defines the structure type segmt, declares the structure variable **thing1**, and declares **papas** which is a pointer to an array of pointers to arrays of structures of type segmt! (Note that the brackets do not contain numeric subscripts because **papas** is a pointer variable.)

Contemplation of derived variable declarations or template defintions will reveal the reason why they are referred to as derived. The last logical reference in these statements must always be to a fundamental data type or function. For instance, an array of arrays is not allowed, but an array of arrays containing int's is.

Index