Programming in C[1]

Bharat Kinariwala
University of Hawai‘i

Tep Dobry
University of Hawai‘i

January 5, 1993

Contents

List of Figures

List of Tables

Preface

The C language has boomed in popularity and availability since its creation in the 70's. It has largely become the language of choice for systems programming as well as general purpose programming in both the numeric and symbolic realms. As a result, all programmers today should have some working knowledge of C, particularly in engineering.

This book is intended to be a first text in programming in general with emphasis on the C language. It is meant for students with little or no previous programming experience and as such, a primary focus is on the top down *design* of programs, beginning with the development of an algorithm, proceeding to the translation of the algorithm into a programming language (C), and the subsequent testing and debugging of the resulting code. Throughout the text, emphasis is placed on organization and readability of code as well as debugging aids in program development. In addition, understandable and functional user interfaces are described.

As an introductory text on programming, our approach is to motivate the introduction of features of the language through example problems. We start with meaningful but simple tasks, develop an algorithm to solve the task and then introduce the necessary language constructs to implement the algorithm. We then refine the task, adding complexity or desirable features to motivate introduction of new language constructs. As such, the text is not meant to be a C reference manual, but a text on program design utilizing the available language features to implement the design. However, for student's reference, the key constructs introduced are summarized at the end of each chapter. The intent is for the student to be able to design and code programs from the very beginning.

The book is organized as follows:

- Chapter 1 is an introduction to computers and some of the terminology used throughout the text.

- Chapter 2 begins the development of a simple C program and the introduction to the organization and basic statements of the language.

- Chapter 3 stresses the top down approach to design and introduces functions at an early stage to emphasize their relation to algorithms.

- Chapter 4 introduces the character data type and algorithms for processing characters.

- Chapter 5 presents numeric data types and their limitations and discusses the details of C expression evaluation.

- Chapter 6 addresses the important concept of pointers and their use in C in functions.

- Chapter 7 introduces compound data types with arrays and discusses their relation with pointers.

- Chapter 8 describes some of the standard library functions provided in C for character and math processing as well as giving a detailed description of the standard I/O functions printf() and scanf() and their variations for file I/O.

- Chapter 9 presents some standard sorting and searching techniques.

- Chapter 10 describes the powerful string processing utilities in C and the concept of libraries of functions.

- Chapter 11 returns to arrays presenting two dimensional arrays.

- Chapter 12 discusses the remaining compound data type; structures and unions.

- Chapter 13 presents advanced file Input/Output features of the language.

- Chapter 14 describes the memory organization of C programs and discusses the details of storage classes and scope.

- Finally, Chapter 15 provides several examples of algorithms useful in Engineering computation. It makes use of the concepts presented in earlier Chapters and these examples may be discussed with the appropriate Chapter.

In addition, three Appendices are provided as follows:

- Appendix A provides a summary of the C language constructs discussed in the text.

- Appendix B contrasts the language features of ANSI C as presented in this text to "old" C which is still available on many Unix systems.

- Appendix C summarizes the standard library functions available in C.

Chapter 1

Introduction

In our modern society Electronic Digital Computer Systems, commonly referred to as **computer systems** or **computers**, are everywhere. We find them in offices, factories, hospitals, schools, stores, libraries, and now in many homes. Computers show up in sometimes unexpected places – in your car, your television and your microwave, for example. We use computers to perform tasks in science, engineering, medicine, business, government, education, entertainment, and many other human endeavors. Computers are in demand wherever complex and/or high speed tasks are to be performed.

Computers have become indispensable tools of modern society. They work at high speed, are able to handle large amounts of data with great accuracy, and have the ability to carry out a specified sequence of operations, i.e. a **program** without human intervention and are able to change from one program to another on command.

Computer systems are general purpose *information processing machines* used to solve problems. Solving these problems may involve processing information (i.e., **data**) which represent numbers, words, pictures, sounds, and many other abstractions. Because we are talking about digital computers, the information to be processed must be represented as discrete values selected from a (possibly very large but finite) set of individual values. For example, integer numbers (the counting numbers) can be represented in a computer by giving a unique *pattern* to each integer up to the maximum number of patterns available to the particular machine. We will see how these patterns are defined in a later section of this Chapter. This mapping of an internal machine *pattern* to a *meaning* is refered to as a **data type**.

Given a representation of information, we would like to be able to perform operations on this data such as addition or comparison. The fundamental operations provided in a computer are very simple logical and arithmetic operations; however, these simple operations can be combined to perform more complex operations. For example, multiplication can be performed by doing repeated additions. The basic operations provided by a particular computer are called **instructions** and a well defined sequence of these instructions is called a **program**. It is the job of the programmer, then, to represent the information of the problem using the data types provided and to specify the sequence of operations which must be performed to solve the problem. As we will
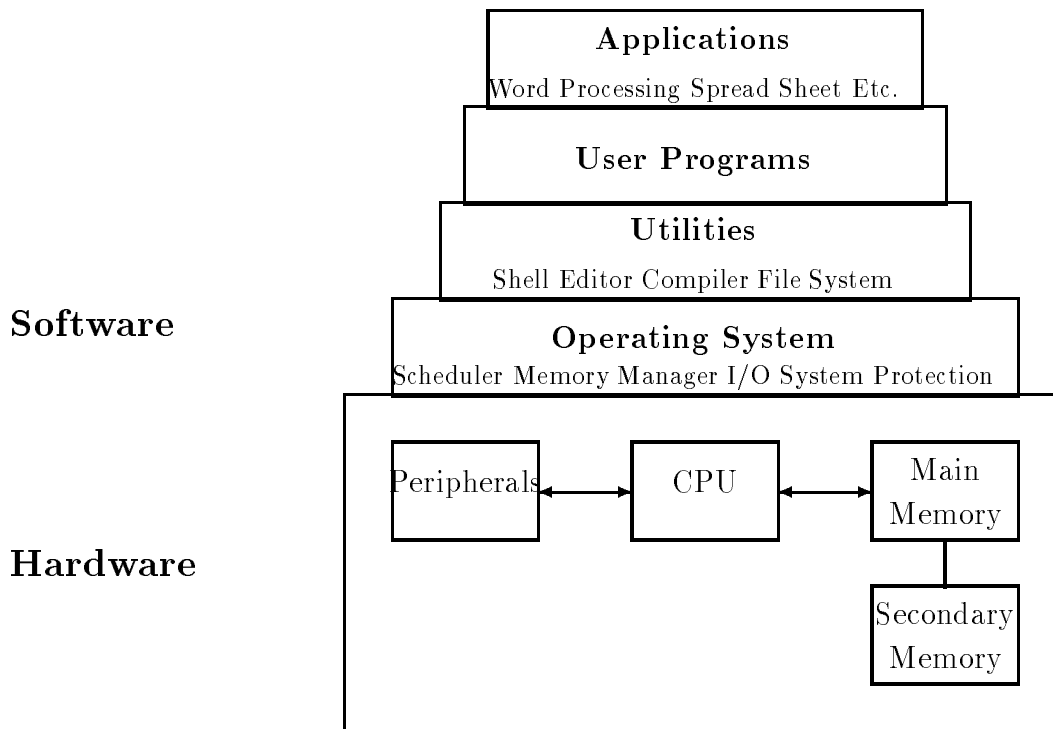
Figure 1.1: Computer System Block Diagram

see in Section 1.2.3, because of the simple nature of the operations available, specifying the proper sequence of instructions to perform a task can be a very complex and tedious task. Fortunately for us, this task has been made simpler these days (using the computers themselves) through the use of high level programming languages. It is one of these languages, the C language that we will discuss in this text.

1.1   Computer System Organization

Before we look at the C language, let us look at the overall organization of computing systems. Figure 1.1 shows a block diagram of a typical computer system. Notice it is divided into two major sections; *hardware* and *software*.

1.1.1   Computer Hardware

The physical machine, consisting of electronic circuits, is called the **hardware**. It consists of several major units: the *Central Processing Unit* (CPU), *Main Memory*, *Secondary Memory* and *Peripherals*.

The CPU is the major component of a computer; the "electronic brain" of the machine. It consists of the electronic circuits needed to perform operations on the data. Main Memory is where programs that are currently being executed as well as their data are stored. The CPU

fetches program instructions in sequence, together with the required data, from Main Memory and then performs the operation specified by the instruction. Information may be both read from and written to any location in Main Memory so the devices used to implement this block are called **random access memory** chips (RAM). The contents of Main Memory (often simply called **memory**) are both temporary (the programs and data reside there only when they are needed) and volatile (the contents are lost when power to the machine is turned off).

The Secondary Memory provides more long term and stable storage for both programs and data. In modern computing systems this Secondary Memory is most often implemented using *rotating magnetic storage devices*, more commmonly called *disks* (though magnetic tape may also be used); therefore, Secondary Memory is often referred to as **the disk**. The physical devices making up Secondary Memory, the *disk drives*, are also known as **mass storage devices** because relatively large amounts of data and many programs may be stored on them.

The disk drives making up Secondary Memory are one form of *Input/Output* (I/O) device since they provide a means for information to be brought into (input) and taken out of (output) the CPU and its memory. Other forms of I/O devices which transfer information between humans and the computer are represented by the *Peripherals* box in Figure 1.1. These Peripherals include of devices such as terminals – a keyboard (and optional mouse) for input and a video screen for output, high-speed printers, and possibly floppy disk drives and tape drives for permanent, removable storage of data and programs. Other I/O devices may include high-speed optical scanners, plotters, multiuser and graphics terminals, networking hardware, etc. In general, these devices provide the *physical interface* between the computer and its environment by allowing humans or even other machines to communicate with the computer.

1.1.2   Computer Software – The Operating System

Hardware is called "hard" because, once it is built, it is relatively difficult to change. However, the hardware of a computer system, by itself, is useless. It must be given directions as to what to do, i.e. a program. These programs are called **software**; "soft" because it is relatively easy to change both the instructions in a particular program as well as which program is being executed by the hardware at any given time. When a computer system is purchased, the hardware comes with a certain amount of software which facilitates the use of the system. Other software to run on the system may be purchased and/or written by the user. Some major vendors of computer systems include: IBM, DEC, HP, AT&T, Sun, Compaq, and Apple.

The remaining blocks in Figure 1.1 are typical software layers provided on most computing systems. This software may be thought of as having a hierarchical, layered structure, where each layer uses the facilities of layers below it. The four major blocks shown in the figure are the *Operating System*, *Utilities*, *User Programs* and *Applications*.

The primary responsibility of the *Operating System* (OS) is to "manage" the "resources" provided by the hardware. Such management includes assigning areas of memory to different programs which are to be run, assigning one particular program to run on the CPU at a time, and controlling the peripheral devices. When a program is called upon to be **executed** (its operations

performed), it must be **loaded**, i.e. moved from disk to an assigned area of memory. The OS may then direct the CPU to begin fetching instructions from this area. Other typical responsibilities of the OS include Secondary Storage management (assignment of space on the disk), a piece of software called the file system, and Security (protecting the programs and data of one user from activities of other users that may be on the same system).

Many mainframe machines normally use proprietary operating systems, such as VM and CMS (IBM) and VAX VMS and TOPS 20 (DEC). More recently, there is a move towards a standardized operating system and most workstations and desktops typically use Unix (AT&T and other versions). A widely used operating system for IBM PC and compatible personal computers is DOS (Microsoft). Apple Macintosh machines are distinguished by an easy to use proprietary operating system with graphical icons.

1.1.3   Utility Programs

The layer above the OS is labeled *Utilities* and consists of several programs which are primarily responsible for the *logical interface* with the user, i.e. the "view" the user has when interacting with the computer. (Sometimes this layer and the OS layer below are considered together as the operating system). Typical utilities include such programs as *shells*, *text editors*, *compilers*, and (sometimes) the *file system*.

A **shell** is a program which serves as the primary interface between the user and the operating system. The shell is a "command interpreter", i.e. is prompts the user to enter commands for tasks which the user wants done, reads and interprets what the user enters, and directs the OS to perform the requested task. Such commands may call for the execution of another utility (such as a text editor or compiler) or a user program or application, the manipulation of the file system, or some system operation such as logging in or out. There are many variations on the types of shells available, from relatively simple command line interpreters (DOS) or more powerful command line interpreters (the Bourne Shell, *sh*, or C Shell, *csh* in the Unix environment), to more complex, but easy to use graphical user interfaces (the Macintosh or Windows). You should become familiar with the particular shell(s) available on the computer you are using, as it will be your primary means of access to the facilities of the machine.

A **text editor** (as opposed to a word processor) is a program for entering programs and data and storing them in the computer. This information is organized as a unit called a **file** similar to a file in an office filing cabinet, only in this case it is stored on the disk. (Word processors are more complex than text editors in that they may automatically format the text, and are more properly considered applications than utilities). There are many text editors available (for example *vi* and *emacs* on Unix systems) and you should familiarize yourself with those available on your system.

As was mentioned earlier, in today's computing environment, most programming is done in high level languages (HLL) such as C. However, as we shall see in Section 1.2.3, the computer hardware cannot understand these languages directly. Instead, the CPU executes programs coded in a lower level language called the **machine language**. A utility called a **compiler** is program which translates the HLL program into a form understandable to the hardware. Again, there are

many variations in compilers provided (for different languages, for example) as well as facilities provided with the compilers (some may have built-in text editors or debugging features). Your system manuals can describe the features available on your system.

Finally, another important utility (or task of the operating system) is to manage the *file system* for users. A file system is a collection of files in which a user keeps programs, data, text material, graphical images, etc. The file system provides a means for the user to organize files, giving them names and gathering them into *directories* (or folders) and to manage their file storage. Typical operations which may be done with files include creating new files, destroying, renaming, and copying files.

1.1.4   User Programs and Applications

Above the utilities in Figure 1.1 is the block labeled *User Programs*. It is at this level where a computer becomes specialized to perform a task to solve a user's problem. Given a task that needs to be performed, a programmer can design and code a program to perform that task using the text editors, compilers, debuggers, etc. The program so written may make use of operating system facilities, for example to do I/O to interact with the program user. It is at this level that the examples, exercises and problems in this text will be written.

However, not everyone who uses a computer is a programmer or desires to be a programmer. As well, if every time a new task was presented to be programmed, one had to start from scratch with a new program, the utility and ease of using the computers would be reduced. These days packages of predefined software, or *Applications*, are available from many vendors in the industry. Highly functional *word processors*, *desktop publishing* packages, *spread sheet* and *data base* programs and, yes, *games* are readily available for computer users as well as programmers. In fact, perhaps most computer users these days access their machines exclusively through these application programs.

A computer system is typically purchased with an operating system, a variety of utilities (such as compilers for high level languages and text editors) and application programs. Without the layers of software in modern computers, computer systems would not be as useful and popular as they are today. While the complexity of these underlying layers has increased greatly in recent years, the net effect has been to make computers easier for people to use.

In the remainder of this Chapter we will take a more detailed look at how data and programs are represented within the machine. We finally discuss the design of programs and their coding in the C language before beginning a detailed description in Chapter 2.

1.2   Representing Data and Program Internally

In a computer, it is the hardware discussed in the previous section that stores data items and programs and that performs operations on these items. This hardware is implemented using electronic circuits called **gates** which, because we are talking about digital computers, represent

information using only two values: *True* and *False*. In most machines, these two values are represented by two different voltages with in the circuit; for example 0 Volts representing a False value, and +5 Volts representing a True value. One such value is called a **binary digit** or **bit** and each such bit can be considered to be a symbol for a piece of information. However, in computer applications we need to represent information that can have more than just two values, i.e. we have more than 2 symbols. So bits are grouped together and the pattern of values on the group is used to represent a symbol. For example, a group of 8 bits, called a **byte** can have 256 different patterns (we will see how below) and therefore represent 256 different symbols. In modern computers, groupings of bytes (usually 2 or 4), called **words** can represent larger "chunks" of information.

Simply representing symbols in a computer, however, is not sufficient. We also need to *process* the information, i.e. perform operations on it. The designers of the hardware make use of an algebra, called **Boolean Algebra**, which uses two values, 0 and 1, and logical operations (AND, OR and NOT) to design the circuits that perform more complex operations on bytes and words of data. These complex operations are the **instruction set** of the computer and are the basic tools the programmer has to write software for the computer. All executable programs must be sequences of instructions from this set which includes basic arithmetic, logical, store and retrieve, and program control instructions. The instructions themselves can also be represented in the machine as patterns of bits.

This section first discusses how different types of data are represented using patterns of bits, then describes how data, as well as instructions, are stored in memory, and finally gives a short example of how instructions are represented.

1.2.1   Representing Data

Standard methods for representing commonly used numeric and non-numeric data have been developed and are widely used. While a knowledge of internal binary representation is not required for programming in C, an understanding of internal data representation is certainly helpful.

Binary Representation of Integers

As mentioned above, all data, including programs, in a computer system is represented in terms of groups of binary digits. A single bit can represent one of two values, 0 or 1. A group of two bits can be used to represent one of four values:

```
        00 --- 0
        01 --- 1
        10 --- 2
        11 --- 3
```

If we have only four symbols to represent, we can make a one-to-one correspondence between the patterns and the symbols, i.e., one and only one symbol is associated with each binary pattern.

For example, the numbers 0, 1, 2, and 3 are mapped to the patterns above.

Such a correspondence is called a **code** and the process of representing a symbol by the corresponding binary pattern is called **coding** or **encoding**. Three binary digits can be used to represent eight possible distinct values using the patterns:

```
000              100
001              101
010              110
011              111
```

A group of k binary digits (bits) can be used to represent 2^k symbols. Thus, 8 bits are used to represent $2^8 = 256$ values, 10 bits to represent $2^{10} = 1024$ values, and so on. It should be clear by now that powers of two play an important role because of the binary representation of all data. The number 1024 is close to one thousand, and it is called $1K$, where K stands for Kilo; nK equals $n * 1024$, and if $n = 2^m$, then nK is $2^{(10+m)}$.

We will first present a standard code for natural numbers, i.e., unsigned integers 0, 1, 2, 3, 4, etc. There are several ways to represent these numbers as groups of bits, the most natural way is analogous to the method we use to represent decimal numbers. Recall, a decimal (or base 10) representation uses exactly ten digit symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Any decimal number is represented using a weighted positional notation.

For example, a single digit number, say 9, represents just nine, because the weight of the rightmost position is 1. A two digit number, say 39, represents thirty plus nine. The rightmost digit has a weight 1, and the next digit to the left has a weight of 10. So, we multiply 3 by 10, and add 9 multiplied by 1. Thus, for decimal notation the weights for the digits starting from the rightmost digit and moving to the left are 1, 10, 100, and so on, as shown below.

5	4	3	2	1	0	digit position
100000	10000	1000	100	10	1	position weight

Thus,

$$3456 = (6 * 1) + (5 * 10) + (4 * 100) + (3 * 1000)$$

The positional weights are the powers of the base value 10, with the rightmost position having the weight of 10^0, the next positions to the left having in succession the weight of 10^1, 10^2, 10^3, and so on. Such an expression is commonly written as a sum of the contribution of each digit, starting with the lowest order digit and working toward the largest weight; that is, as sums of contributions of digits starting from the rightmost position and working toward the left.

Thus, if i is an integer written in decimal form with digits d_k:

$$i = d_{n-1}d_{n-2}\ldots d_2 d_1 d_0$$

then i represents the sum:

$$i = \sum_{k=0}^{n-1} d_k * 10^k$$

where n is the total number of digits, and d_k is the kth digit from the rightmost position in the decimal number.

Binary representation of numbers is written in exactly the same manner. The base is 2, and a number is written using just two digits symbols, 0 and 1. The positional weights, starting from the right are now powers of the base 2. The weight for the rightmost digit is $2^0 = 1$, the next digit has the weight of $2^1 = 2$, the next digit has the weight of $2^2 = 4$, and so on. Thus, the weights for the first ten positions from the right are as follows:

10	9	8	7	6	5	4	3	2	1	0	position
1024	512	256	128	64	32	16	8	4	2	1	pos. weights

A natural binary number is written using these weights. For example, the binary number

$$1\ 0\ 0\ 1\ 0$$

represents the number whose decimal equivalent is

$$2^1 + 2^4 = 2 + 16 = 18$$

and the binary number

$$1\ 0\ 1\ 0\ 1\ 0\ 0\ 0$$

represents the number whose decimal equivalent is

$$2^3 + 2^5 + 2^7 = 8 + 32 + 128 = 168$$

When a binary number is stored in a computer word with a fixed number of bits, unused bits to the left (leading bits) are set to 0. For example, with a 16 bit word, the binary equivalent of 168 is

$$0000\ 0000\ 1010\ 1000$$

We have shown the bits in groups of four to make it easier to read.

In general, if i is an integer written in binary form with digits b_k:

$$i = b_{n-1}b_{n-2}\ldots b_2 b_1 b_0$$

then its decimal equivalent is:

$$i = \sum_{k=0}^{n-1} b_k * 2^k$$

As we said, a word size of k bits can represent 2^k distinct patterns. We use these patterns to represent the unsigned integers from 0 to $2^k - 1$. For example, 4 bits have 16 distinct patterns representing the equivalent decimal unsigned integers 0 to 15, 8 bits for decimal numbers 0 through 255, and so forth.

Given this representation, we can perform operations on these unsigned integers. Addition of two binary numbers is straightforward. The following examples illustrate additions of two single bit binary numbers.

```
    0              0              1              1
   +0             +1             +0             +1
   ---            ---            ---            ---
    0              1              1             10
```

The last addition, $1 + 1$, results in a sum digit of 0 and a carry forward of 1. Similarly, we can add two arbitrary binary numbers, b1 and b2:

```
            011100        (carry forward    )

   b1       101110        (base 10 value: 46)
  +b2      +001011        (base 10 value: 11)
  ---      --------
  sum       111001        (base 10 value: 57)
```

Decimal to Binary Conversion

We have seen how, given a binary representation of a number, we can determine the decimal equivalent. We would also like to go the other way; given a decimal number, find the corresponding binary pit pattern representing this number. In general, there are two approaches; one generates the bits from the most significant (the leftmost bit) to the least significant; the other begins with the rightmost bit and proceeds to the leftmost.

In the first case, to convert a decimal number, n, to a binary number, determine the highest power, k, of 2 that can be subtracted from n:

$$r = n - 2^k$$

and place a 1 in the kth binary digit position. The process is repeated for the remainder r, and so forth until the remainder is zero. All other binary digit positions have a zero. For example, consider a decimal number 103. The largest power of 2 less than 103 is 64 (2^6):

$$
\begin{aligned}
103 - 2^6 &= 103 - 64 = 39 \\
39 - 2^5 &= 39 - 32 = 7 \\
7 - 2^2 &= 7 - 4 = 3 \\
3 - 2^1 &= 3 - 2 = 1 \\
1 - 2^0 &= 1 - 1 = 0
\end{aligned}
$$

So we get:

```
weights        128  64   32   16   8    4    2    1
                     1    1              1    1    1
```

which, for an 8 bit representation give:

0110 0111

In the alternate method, we divide n by 2, using integer division (discarding any fractional part), and the remainder is the next binary digit moving from least significant to most. In the example below, the % operation is called **mod** and is the remainder from integer division.

$$
\begin{aligned}
103 \ \% \ 2 &= 1 & 103 \ / \ 2 &= 51 \\
51 \ \% \ 2 &= 1 & 51 \ / \ 2 &= 25 \\
25 \ \% \ 2 &= 1 & 25 \ / \ 2 &= 12 \\
12 \ \% \ 2 &= 0 & 12 \ / \ 2 &= 6 \\
6 \ \% \ 2 &= 0 & 6 \ / \ 2 &= 3 \\
3 \ \% \ 2 &= 1 & 3 \ / \ 2 &= 1 \\
1 \ \% \ 2 &= 1 & 1 \ / \ 2 &= 0
\end{aligned}
$$

Reading the bits top to bottom filling right to left, the number is

0110 0111

Representing Signed Integers

The binary representation discussed above is a standard code for storing unsigned integer numbers. However, most computer applications use signed integers as well; i.e. integers that may be either positive or negative. There are several methods used for representing signed numbers.

The first, and most obvious, is to represent signed numbers as we do in decimal, with an indicator for the sign followed by the magnitude of the number as an unsigned quantity. For example, we write:

+100

−100

In binary we can use one bit within a representation (usually the most significant or leading bit) to indicate either positive (0) or negative (1), and store the unsigned binary representation of the magnitude in the remaining bits. So for an 16 bit word, we can represent the above numbers as:

$$+100: \quad 0000\ 0000\ 0110\ 0100$$

$$-100: \quad 1000\ 0000\ 0110\ 0100$$

However; for reasons of ease of design of circuits to do arithmetic on signed binary numbers (e.g. addition and subtraction), a more common representation scheme is used called **two's complement**. In this scheme, positive numbers are represented in binary, the same as for unsigned numbers. On the other hand, a negative number is represented by taking the binary representation of the magnitude, complementing all bits (changing 0's to 1's and 1's to 0's), and adding 1 to the result.

Let us examine the 2's complement representation of +100 and −100 using 16 bits. For +100, the result is the same as for unsigned numbers:

$$+100: \quad 0000\ 0000\ 0110\ 0100$$

For −100, we begin with the unsigned representation of 100:

$$0000\ 0000\ 0110\ 0100$$

complement each bit:

$$1111\ 1111\ 1001\ 1011$$

and add 1 to the above to get 2's complement:

$$-100: \quad 1111\ 1111\ 1001\ 1100$$

This operation is reversible, that is, the magnitude (or absolute value) of a two's complement representation of a negative number can be obtained with the same procedure; complement all bits:

$$0000\ 0000\ 0110\ 0011$$

and add 1:

$$0000\ 0000\ 0110\ 0100$$

In a two's complement representation, we can still use the most significant bit to determine the sign of the number; 0 for positive, and 1 for negative. Let us determine the decimal value of a negative 2's complement number:

$$1111\ 1111\ 1101\ 0110$$

This is a negative integer since the leading bit is 1, so to find its magnitude we complement all bits:.

$$0000\ 0000\ 0010\ 1001$$

and add 1:

$$0000\ 0000\ 0010\ 1010$$

The decimal magnitude is 42, and the sign is negative, so, the original integer represents decimal -42.

In determining the range of integers that can be represented by k bits, we must allow for the sign bit. Only $k - 1$ bits are available for positive integers, and the range for them is 0 through $2^{(k-1)} - 1$. The range of negative integers representable by k bits is -1 through $-2^{(k-1)}$. Thus, the range of integers representable by k bits is $-2^{(k-1)}$ through $2^{(k-1)} - 1$. For example, for 8 bits, the range of signed integers is $-2^{(8-1)}$ through $2^{(8-1)} - 1$, or -128 to $+127$.

It can be seen from the above analysis that, due to a finite number of bits used to represent numbers, there are limits to the largest and/or smallest numbers that can be represented in the computer. We will discuss this further in Chapter 5.

Octal and Hexadecimal Representations

One important thing to keep in mind at this point is that we have been discussing different *representations* for numbers. Whether a number is expressed in binary, e.g. 010011, or decimal, 19, it is still the same number, namely nineteen. It is simply more convenient for people to think in decimal and for the computer to use binary. However, converting the computer binary representation to the human decimal notation is somewhat tedious, but at the same time writing long strings of bits is also inconvenient and error prone. So two other representation schemes are commonly used in working with binary representations. These schemes are call **octal** and **hexadecimal** (sometimes called **hex**) representations and are simply positional number systems using base 8 and 16, respectively.

In general, an unsigned integer, i, consisting of n digits d_i written as:

$$i = d_{n-1} d_{n-2} \ldots d_3 d_2 d_1 d_0$$

in any base is interpreted as the sum:

$$i = \sum_{k=0}^{n-1} d_k * \mathrm{base}^k$$

If the base is 2 (binary), the symbols which may be used for the digits d_i are [0, 1]. If the base is 10 (decimal) the digit symbols are [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Likewise, for base 8 (octal) the digit symbols are [0, 1, 2, 3, 4, 5, 6, 7]; and for hexadecimal (base 16) they are [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f]. The letter symbols, [a, b, c, d, e, f] (upper case [A, B, C, D, E, F] may also be used) give us the required 16 symbols and correspond to decimal values [10, 11, 12, 13, 14, 15] respectively. Using the above sum, it should be clear that the following are representations for the same number:

Base	10:	423
Base	2:	0000 0001 1010 0111
Base	8:	000647
Base	16:	01A7

For hexadecimal numbers, the positional weights are, starting from the right, 1, 16, 256, etc. Here are a few examples of converting hex to decimal:

Hexadecimal	Decimal		
30	0 * 1 + 3 * 16	=	48
1E	14 * 1 + 1 * 16	=	30
1C2	2 * 1 + 12 * 16 + 1 * 256	=	450

Similarly, octal numbers are base 8 numbers with weights 1, 8, 64, etc. The following are some examples of converting octal to decimal:

Octal	Decimal		
11	1 * 1 + 1 * 8	=	9
20	0 * 1 + 2 * 8	=	16
257	7 * 1 + 5 * 8 + 2 * 64	=	175

The reason octal and hex are so common in programming is the ease of converting between these representations and binary, and vice versa. For hexadecimal numbers, exactly four bits are needed to represent the symbols 0 through F. Thus, segmenting any binary number into 4 bit groups starting from the right, and converting each group to its hexadecimal equivalent gives the hexadecimal representation.

```
Binary:      1010        1000
Hex:          A            8
            10 * 16   +   8 * 1
Decimal:          168
```

As a side effect, conversion from binary to decimal is much easier by first converting to hex and then to decimal, as shown above.

Similarly, segmenting a binary number into three bit groups starting from the right gives us the octal representation. Thus, the same number can be expressed in octal as:

```
Binary:     10       101        000
Octal:       2         5          0
          2 * 64   +  5 * 8   +  0 * 1
Decimal:          168
```

Conversion of base 8 or base 16 numbers to binary is very simple; for each digit, its binary representation is written down. Conversion between octal and hex is easiest done by converting to binary first:

Decimal	122	199	21	63
Binary	01111010	11000111	010101	111111
Hexadec.	0111 1010 0X7A	1100 0111 0XC7	0001 0101 0X15	0011 1111 0X3F
Octal	01 111 010 0172	11 000 111 0307	00 010 101 025	00 111 111 077

Table 1.1: Number Representations

Hex:	2	f	3	
Binary:	0010	1111	0011	

Binary:	001	011	110	011
Octal:	1	3	6	3

Some additional examples of equivalent hexadecimal, octal, binary, and decimal numbers are shown in Table 1.1 In a programming language we need a way to distinguish numbers written in different bases (base 8, 16, 10, or 2). In C source programs, a simple convention is used to write constants in different bases. Decimal numbers are written without leading zeros. Octal numbers are written with a leading zero, e.g. 0250 is octal 250. Hexadecimal numbers are written with a leading zero followed by an x or X, followed by the hexadecimal digits. Thus, 0xA8 will mean hexadecimal A8. (Binary numbers are not used in source programs).

Representing Other Types of Data

So far we have discussed representations of integers, signed and unsigned; however, many applications make use of other types of data in their processing. In addition, some applications using integers require numbers larger than can be stored in the available number of bits. To address these problems, another representation scheme, called **floating point** is used. This scheme allows representation of numbers with fractional parts (real numbers) as well as numbers that may be very large or very small.

Representation of floating point numbers is analogous to decimal *scientific notation*. For example:

$$1.234 * 10 + 2$$

$$.1234 * 10 + 3$$

By adjusting the decimal place, as in the last case above, a number of this form consists of just three parts: a fractional part called the **mantissa**, a base, and an exponent. Over the years, several schemes have been devised for representing each of these parts and storing them as bits in a computer. However, in recent years a standard has been defined by the Institute for Electrical and Electronics Engineers (IEEE Standard 754) which is gaining in acceptance and use in many computers. A detailed description of these schemes, and their relative tradeoffs, is beyond the scope of this text; however, as programmers, it is sufficient that we realize that we can express floating point numbers either in decimal with a fractional part:

$$325.54927$$

or using exponential form:

$$3.2554927E + 2$$

$$325549.27E - 3$$

where E or e refers to exponent of the base (10 in this case). As with integers, due to the fixed number of bits used in the representation, there are limits to the range (largest and smallest numbers) and the precision (number of digits of accuracy) for the numbers represented.

Another widely used data type is character data which is non-numeric and includes all the symbols normally used to represent textual material such as letters, digits and punctuation. Chapter 4 discusses the representation of character data in detail, however, the principle is the same; some pattern of bits is selected to represent each symbol to be stored.

These are the principle data types provided by programming languages, but as we will see in future Chapters, languages also provide a means for the programmer to define their own data types and storage schemes.

1.2.2   Main Memory

Now that we have seen that information (data) can be represented in a computer using binary patterns, we can look at how this information is stored within the machine. An electronic circuit that can be switched ON or OFF can represent one binary digit or one bit of information. A class of such devices (called **flip-flops**) which can retain the value of a bit, even after the input to them changes (though only as long as power is applied to them), can be used to store a bit. The Main Memory block of Figure 1.1 is constructed of many of these devices, organized so that data (and instructions) may be stored there and subsequently accessed. Memory in present day computers is usually organized as a sequence of bytes (a **byte** is a group of eight bits). Each byte in memory is given a unique unsigned integer *address*, which may be considered its "name". (See Figure 1.2). A row of houses on a street with street addresses or a row of numbered mailboxes are reasonable analogies to memory addresses. The CPU (or any other device wishing to access memory) may place an address on a set of wires connected to the memory (called the **address bus**) in order to either read (load) or write (store) information in memory. Once information

Binary	Hex	
0000 0000 0000 0000	0000	
0000 0000 0000 0001	0001	
0000 0000 0000 0010	0002	
0000 0000 0000 0011	0003	
0000 0000 0000 0100	0004	
0000 0000 0000 0101	0005	
		. . .
0000 0000 0100 1001	0049	
0000 0000 0100 1010	004A	
0000 0000 0100 1011	004B	
		. . .
1111 1111 1111 1111	FFFF	

Binary   Hex   Memory
Address   Bytes

Figure 1.2: Memory and Addresses

has been written to a particular location (address) in memory, it will remain unchanged unless a subsequent write is performed to the same address. Multiple bytes may be accessed (either simultaneously or sequentially) for data items large than a single byte. Like other information in the computer, an address is represented internally in binary. In the figure, we have shown the addresses both in binary and in hexadecimal form.

Computers are often classified by how many bits may be accessed simultaneously, e.g. 16 bits or 32 bits. The maximum number of bytes directly addressable in a computer depends on the number of bits in the memory address. A 16 bit machine usually allows 16 bits for address and a 32 bit machine usually allows anywhere form 17 to 32 bits for address. Since n bits can represent 2^n values, 16 bit addresses can address 64 KBytes (i.e. 65,536 bytes from byte addresses 0 to 65535) and 32 bit addresses can address 4 GigaBytes (over 4,000,000,000 bytes) directly.

1.2.3   Representing Programs

As has been mentioned, in addition to data being stored in memory, the program to be executed is also stored there in the form of a sequence of instructions. It is the CPU shown in Figure 1.1 that is responsible for fetching instructions, one at a time, from memory and performing the specified operation on data. A more detailed picture of the CPU with its memory is shown in Figure 1.3. Within the CPU are several key components; the ALU, a set of Registers, and a Control Unit.

The ALU (Arithmetic Logic Unit) is a digital circuit which is designed to perform arithmetic

Figure 1.3: CPU and Memory Configuration

(add, subtract) operations as well as logic (AND, OR) operations on data. The registers in the CPU are a small scratchpad memory to temporarily store data while it is in use. The Control Unit is another circuit which determines what operation is being requested by an instruction and controls the other circuitry to carry out that operation; i.e. the Control Unit directs all operations within the machine.

Also shown in the figure are the connections between the CPU and Memory. They consist of an address bus, as mentioned in the previous Section, and a data bus, over which all information (data and program) passes between the CPU and Memory.

This Section describes how programs are stored in the machine as a sequence of instructions coded in binary. Such an encoding is called the **machine language** of the computer and is described below.

Machine Language

The basic operations that the CPU is capable of performing are usually quite simple and the set of these operations provided on a particular computer is called the **instruction set**. Within this set are instructions which can move data from one place to another, for example from memory to a CPU register; an operation called **load**. Similarly there are **store** instructions for moving data from the CPU to a location in memory. In addition there are instructions directing arithmetic

operations, such as add, on data values. There are also instructions which control the flow of the program; i.e. that determine from where in memory the next instruction should be fetched. Normally instructions are fetched sequentially – the next instruction is fetch from the next memory address; however, these control instructions may test a condition and direct that the next instruction be fetched from somewhere else in memory instead. Finally, there may also be instructions in the set for "housekeeping" operations within the machine, such as controlling external I/O devices.

To encode these instructions in binary form for storage in memory, some convention must be adopted to describe the meaning of the bits in the instruction. Most of the instructions described above require at least 2 pieces of information – a specification of what particular instruction this is, called the **opcode** or operation code, and the address of the data item on which to operate. These parts can be seen in Figure 1.3 in the block labeled *instruction*.

Instructions coded in binary form are called **machine language instructions** and the collection of these instructions that make up a program is called a **machine language program**. Such a program is very difficult for a person to understand or to write. Just imagine thinking in terms of binary codes for very low level instructions and in terms of binary memory addresses for data items. It is not practical to do so except for very trivial programs. Humans require a higher level of programming languages that are more adapted to our way of thinking and communicating. Therefore, at a level a little higher than machine language, is a programming language called **assembly language** which is very close to machine language. Each assembly instruction translates to one machine language instruction. The main advantage is that the instructions and memory cells are not in binary form; they have names. Assembly instructions include operational codes, (i.e., mnemonic or memory aiding names for instructions), and they may also include addresses of data. An example of a very simple program fragment for the machine described above is shown in Figure 1.4. The figure shows the machine language code and its corresponding assembly language code. Definitions of memory cells are shown below the program fragment.

The machine language code is shown in binary. It consists of 8 bits of opcode and 16 bits of address for each instruction. From the assembly language code it is a little easier to see what this program does. The first instruction loads the data stored in memory at a location known as "Y" into the CPU register (for CPU's with only one register, this is often called the **accumulator**). The second instruction adds the data stored in memory at location "X" to the data in the accumulator, and stores the sum back in the accumulator. Finally, the value in the accumulator is stored back to memory at location "Y". With the data values shown in memory in the figure, at the end of this program fragment, the location known as "Y" will contain the value 48.

A utility program is provided to translate the assembly language code (arguably) readable by people into the machine language code readable by the CPU. This program is called the **assembler**. The program in the assembly language or any other higher language is called the **source program**, whereas the program assembled into machine language is called the **object program**. The terms source code and object code are also used to refer to source and object programs.

Assembly language is a decided improvement over programming in machine language, however, we are still stuck with having to manipulate data in very simple steps such as load, store, add, etc., which can be a tedious, error prone process. Fortunately for us, programming languages at

```
Program Fragment:        Y = Y + X

Machine Language Code                          Assembly Language
(Binary Code)                                         Code

Opcode          Address
1100 0000       0010 0000 0000 0000              LOAD  Y
1011 0000       0001 0000 0000 0000              ADD   X
1001 0000       0010 0000 0000 0000              STORE Y


Memory Cell Definitions:

     Addr.      Name     Cell Contents

     1000       X        32
     2000       Y        16
```

Figure 1.4: Machine and Assembly Language Program Fragment

higher levels still, languages closer to the way we think about programming, have been developed along with translators (called **compilers**) for converting to object programs. One such language is C, which is the subject of this text and is introduced in the next Section.

1.3   Designing Programs and the C Language

We defined a program as an organized set of instructions stating the steps to be performed by a computer to accomplish a task. **Computer programming** is the process of planning, implementing, testing, and revising (if necessary) the sequences of instructions in order to develop successful programs. In writing computer programs we must specify with precise, unambiguous instructions exactly what we want done and the order in which it should be done. Before we can write the actual program, we must either know or develop a step-by-step procedure, or *algorithm*, that will accomplish the task. We can then implement the algorithm by coding it into a source language program.

1.3.1   Designing The Algorithm

An algorithm is a general solution of a problem which can be written as a verbal description of a precise, logical sequence of actions. Cooking recipes, assembly instructions for appliances and
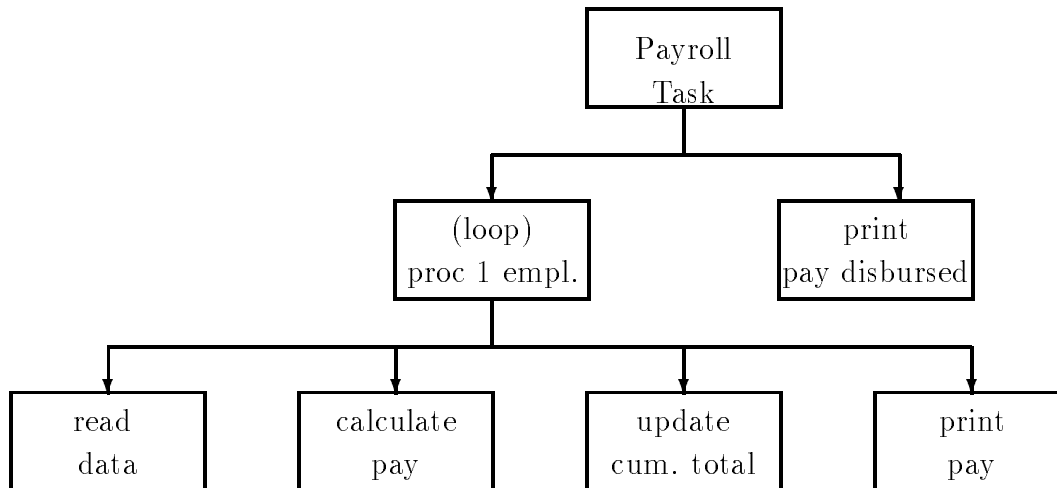
Figure 1.5: Structural Diagram for Payroll Task

toys, or precise directions to reach a friend's house, are all examples of algorithms. A computer program is an algorithm expressed in a specific programming language. An algorithm is the key to developing a successful program.

Suppose a business office requires a program for computing its payroll. There are several people employed. They work regular hours, and sometimes overtime. The task is to compute pay for each person as well as compute the total pay disbursed.

Given the problem, we may wish to express our recipe or *algorithm* for solving the payroll problem in terms of repeated computations of total pay for several people. The logical modules involved are easy to see.

```
Algorithm: PAYROLL

    Repeat the following while there is more data:
        get data for an individual,
        calculate the pay for the individual from the current data,
            and, update the cumulative pay disbursed so far,
        print the pay for the individual.
    After the data is exhausted, print the total pay disbursed.
```

Figure 1.5 shows a **structural diagram** for our task. This is a layered diagram showing the development of the steps to be performed to solve the task. Each box corresponds to some subtask which must be performed. On each layer, it is read from left to right to determine the performance order. Proceeding down one layer corresponds to breaking a task up into smaller component steps – a refinement of the algorithm. In our example, the payroll task is at the top and that box represents the entire solution to the problem. On the second layer, we have divided the problem into two subtasks; processing a single employee's pay in a loop (to be described below), and
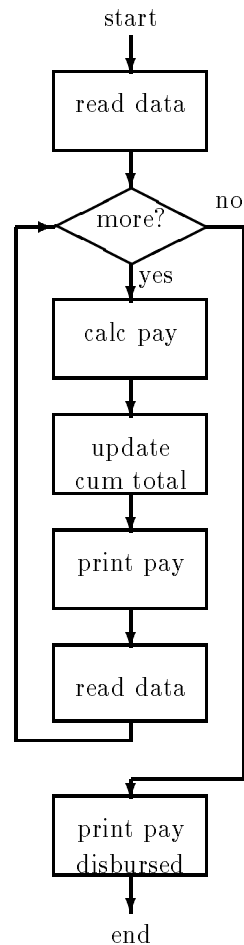
Figure 1.6: Flow Chart for Payroll Task

printing the total pay disbursed for all employees. The subtask of processing an individual pay record is then further refined in the next layer. It consists of, first reading data for the employee, then calculating the pay, updating a cumulative total of pay disbursed, and finally printing the pay for the employee being processed.

The structural diagram is useful in developing the steps involved in designing the algorithm. Boxes are refined until the steps within the box are "doable". Our diagram corresponds well with the algorithm developed above. However, this type of diagram is not very good at expressing the sequencing of steps in the algorithm. For example, the concept of looping over many employees is lost in the bottom layer of the diagram. Another diagram, called a **flow chart** is useful for showing the *control flow* of the algorithm, and can be seen in Figure 1.6. Here the actual flow of control for repetitions is shown explicitly. We first read data since the control flow requires us to test if there is more data. If the answer is "yes" we proceed to the calculation of pay for an individual, updating of total disbursed pay so far, and printing of the individual pay. We then read the next set of data and loop back to the test. If there is more data, repeat the process, otherwise control passes to the printing of total disbursed pay and the program ends.

From this diagram we can write our refined algorithm as shown below. However, one module may require further attention; the one that calculates pay. Each calculation of pay may involve arithmetic expressions such as multiplying hours worked by the rate of pay. It may also involve branching to alternate computations if the hours worked indicate overtime work. Incorporating these specifics, our algorithm may be written as follows:

```
Algorithm: PAYROLL

get (first) data, e.g., id, hours worked, rate of pay
while more data (repeat the following)
     if hours worked exceeds 40
          (then) calculate pay using overtime pay calculation
     otherwise calculate pay using regular pay calculation
     calculate cumulative pay disbursed so far
     print the pay statement for this set of data
     get (next) data

print cumulative pay disbursed
```

The algorithm is the most important part of solving difficult problems. Structural diagrams and flow charts are tools that make the job of writing the algorithm easier, especially in complex programs. The final refined algorithm should use the same type of constructs as most programming languages. Once an algorithm is developed, the job of writing a program in a computer language is relatively easy; a simple translation of the algorithm steps into the proper statements for the language. In this text, we will use algorithms to specify how tasks will be performed. Programs that follow the algorithmic logic will then be easy to implement. Readers may wish to draw structural diagrams and flow charts as visual aids in understanding complex algorithms.

There is a common set of programming constructs provided by most languages useful for algorithm construction, including:

- *Branching*: test a condition, and specify steps to perform for the case when the condition is satisfied (True), and (optionally) when the condition is not satisfied (False). This construct was used in our algorithm as:

```
     if overtime hours exceed 40
          then calculate pay using overtime pay calculation
     otherwise calculate pay using regular pay calculation
```

- *Looping*: repeat a set of steps as long as some condition is True, as seen in:

```
     while new data repeat the following
          ...
```

- *Read* or *print* data from/to peripheral devices. Reading of data by programs is called data input and writing by programs is called data output. The following steps were used in our algorithm:

```
read data
write/print data, individual pay, disbursed pay
```

Languages that include the above types of constructions are called **algorithmic languages** and include such languages as C, Pascal, and FORTRAN.

A program written in an algorithmic language must, of course, be translated into machine language. A Utility program, called a **compiler**, translates source programs in algorithmic languages to object programs in machine language. One instruction in an algorithmic language, called a **statement**, usually translates to several machine level instructions. The work of the compiler, the translation process, is called **compilation**.

To summarize, program writing requires first formulating the underlying algorithm that will solve a particular problem. The algorithm is then coded into an algorithmic language by the programmer, compiled by the compiler, and loaded into memory by the operating system. Finally, the program is executed by the hardware.

1.3.2   The C Language

In this text, our language of choice for implementing algorithms is C. C was originally developed on a small machine (PDP-11) by Dennis Ritchie for implementing the UNIX operating system at Bell Laboratories in Murray Hill, New Jersey (1971-73). C is now used for a wide range of applications including UNIX implementations, systems programming, scientific and engineering computation, spreadsheets, and word processing. In fact, the popularity of C has encouraged the development of a C standard by the American National Institute of Standards (ANSI). This text adheres to *ANSI C*. Major differences between ANSI C and "old C" are pointed out in Appendix B. References at the end of this chapter include books by Kernighan and Ritchie [1, 2], which define both traditional C and ANSI C as well as a reference to the proposed ANSI C standard by Harbison and Steele[3].

In keeping with the original intent, C is a small language; however, it features modern control flow and data structures and a rich set of operators. C provides a wealth of constructs, or statements, which correspond to good algorithmic structures. C uses a standard library of functions to perform many routine tasks such as input and output and string operations. Since C is oriented towards the use of a library of functions, programs in C tend to be modular with numerous small functional modules. It is also possible for users to develop their own libraries of functions to improve program development.

C is fairly standard; programs written in C are easily moved from one machine to another. Such portability of programs is a major advantage in that applications developed on one computer can be

used elsewhere. This allows one to write clear and algorithmically well structured programs. Such a *structured programming* approach is very important in developing complex, error-free applications.

C provides low level logic operations, normally available only in machine language or assembly language. Low level operations are required for *systems programming*, such as writing operating systems and other programs at the system level. Today, many operating systems are written in C. C is also suitable for writing scientific and engineering programs, for example it provides double precision computations of real numbers, as well as long integer computation which can be useful in many applications where a large range of integers is required.

As a first programming language C has some weaknesses; however, they can be overcome by discipline in writing programs. In the text, we will indicate items that beginning programmers need to watch out for.

1.4 Summary

In this Chapter we have given a brief overview of modern computing systems, including both the hardware and software. We had described how information is represented in these machines, both data and programs. We have discussed the development of algorithms as the first, and probably most important step in writing a program. As we shall see, programming is a design process; an algorithm is written, coded, and tested followed by *iteration*. Programs are not written in one step – initial versions are developed and then refined and improved.

One brief note about the organization of chapters in the text. In this chapter (following the References) are two sections labeled *Exercises* and *Problems*. These are very important sections in learning to program, because the only way to learn and improve programming skills is to program. The exercises are designed to be done with pencil-and-paper. They test the key concepts and language constructs presented in the chapter. The problems are generally meant to be computer exercises. They present problems for which programs should be written. By writing these programs you will increase your experience in the methods and thought processes that go into developing ever more complex applications.

With the background of this Chapter, we are ready to begin looking at the specifics of the C language, so

<div align="center">

E hoʻomaka kākou.

(Let's start).

</div>

1.5   References

[1] Kernighan, Brian W., and Ritchie, Dennis M. The C Programming Language, First Edition, Englewood Cliffs, N.J.: Prentice-Hall, 1978.

[2] Kernighan, Brian W., and Ritchie, Dennis M. The C Programming Language, Second Edition, Englewood Cliffs, N.J.: Prentice-Hall, 1988.

[3] Harbison, Samuel P., and Steele, Guy L. Jr., C: A Reference Manual, Second Edition, Englewood Cliffs, N.J.: Prentice-Hall, 1987.

1.6   Exercises

1. Convert the following binary numbers into decimal values:

   ```
   0000 0100 0110 1001
   0011 0001 0111 1111
   0101 0101 0101 0101
   ```

2. Convert the following octal numbers into decimal:

   ```
   000345
   000111
   000777
   ```

3. Convert the following hexadecimal numbers into decimal:

   ```
   1A
   FF
   21
   ```

4. Convert the following decimal integer values into binary, octal, and hexadecimal:

   ```
   101
   324
   129
   ```

5. Add the following binary numbers:

   ```
   0000 0100 0110 1001
   0011 0001 0111 1111
   0101 0101 0101 0101
   ```

6. Add the following octal numbers:

   ```
   000345
   000111
   000777
   ```

7. Add the following hexadecimal numbers:

   ```
   1A
   FF
   21
   ```

8. How many distinct binary strings can be formed with n bits?

9. Find the negative of the following binary numbers in a two's complement representation:

    ```
    0000 0100 0110 1001
    0011 0001 0111 1111
    0101 0101 0101 0101
    ```

10. Represent the following in two's complement form using 16 bits:

    ```
    -29
     165
    -100
    ```

11. What is the largest positive integer that can be stored in n bits, with one leading bit reserved for the sign bit? Explain. Negative integer? Assume two's complement representations.

1.7   Problems

1. Develop an algorithm for the calculation of the value of each stock and the total value of a portfolio of stocks. Draw a structural diagram and write the algorithm using constructions used in the text.

2. Develop an algorithm for calculating and printing the squares of a set of numbers. Draw a structural diagram, a flow chart, and write the algorithm.

3. Develop an algorithm for calculation of the grade point ratio for each student, i.e., (total grade points) / (total credit hours). Each student earns grades (0-4) in a set of courses, each course with different credit hours (1-3). Grade points in one course are given by the product of the grade and the credit hours for the course. Draw a structural diagram and a flow chart.

4. Assume that an "add" operator is available, but not a "multiply" operator in a programming language. Develop an algorithm that will multiply two positive integers using only the "add" operator.

5. Assume that you are only able to read the numeric value of each successive digits of a decimal integer one digit at a time. The objective is to find the overall numeric value of the number. As each new digit is read, the overall numeric equivalent must be updated to allow for the new digit. For example, if the digits read are 3,2, and 5, the result printed should be 325. Extend the algorithm for a number in any specified base.

6. Log in to the computer system available to you. Practice using the text editor available by entering the following simple program and storing it in a file:

```
main()
{
printf("hello world\n");
}
```

7. Compile the program you entered in Problem 6. Note which file have been created during compilation. Execute the compiled program.

8. Explore the computer you will be using. See what applications may be available to you such as electronic mail, and news.

Chapter 2

Basic Concepts

Learning to program is a lot like learning to speak a new language. You must learn new *vocabulary*, i.e. the *words* of the language; the *syntax*, (also called the **grammar**); i.e. the form of statements in the language, as well as the *semantics*, i.e. the *meaning* of the words and statements. This learning process usually begins slowly but often you find that with just a few basic words and phrases you can begin conversing and getting your thoughts across. In this chapter we present a few of the basic statements of the C language so that you can write programs from the beginning.

As in spoken languages, the first thing you need is something to say – an idea. In the programming world, this idea is often in the form of a *task*, i.e. something you would like to have done by the computer. The task may be described in terms of what information is to be provided to the computer, what is to be done with this information, and what results should be produced by the program. A program is often developed in small increments, starting with a relatively simple version of the task and progressing to more complex ones, adding features until the entire task can be solved. The focus is always on the task to be performed. The task must be clearly understood in order to proceed to the next step, the development of an *algorithm*. As was discussed in the previous chapter, an **algorithm** is a step by step description of what must be done to accomplish a task. These can be considered to be the most important steps in programming; specifying and understanding the task (what is to be done), and designing the algorithm (how it is to be done). We take this approach beginning in this chapter, and we will discuss task development and algorithm design in more detail in Chapter 3.

Once an algorithm is clearly stated, the next step is to translate the algorithm into a *programming language*. In our case this will be the C language. Using the vocabulary, syntax, and semantics of the language, we can *code* the program to carry out the steps in the algorithm. After coding a program, we must test it by running it on the computer to ensure that the desired task is indeed performed correctly. If there are **bugs**, i.e. errors in the program, they must be removed; in other words an erroneous program must be *debugged* so it performs correctly. The job of programming includes the entire process: algorithm development, and coding, testing and debugging the program.

At the end of the Chapter, you should know:

- How to code simple programs in C.

- How a program allocates memory to store data, called **variables**.

- How variables are used to store and retrieve data, and to make numeric calculations.

- How decisions are made based on certain events, and how a program can branch to different paths.

- How a set of computations can be repeated any number of times.

- How a program can be tested for errors and how the errors may be removed.

2.1   A Simple C Program

The easiest way to learn programming is to take simple tasks and see how programs are developed to perform them. In this section we will present present one such program explaining what it does and showing how it executes. A detailed description of the syntax of the statments used is given in Section 2.2.

2.1.1   Developing the Algorithm

In the previous chapter we introduced a payroll task which can be summarized as a task to calculate pay for a number of people employed by a company. Let us assume that each employee is identified by an id number and that his/her pay is computed in terms of an hourly rate of pay. We will start with a simple version of this task and progress to more complex versions. The simplest version of our task can be stated as follows.

Task

PAY0: Given the hours worked and rate of pay, write a program to compute the pay for a person with a specified id number. Print out the data and the pay.

The algorithm in this case is very simple:

```
print title of program;
set the data: set id number, hours worked, and rate of pay;
set pay to the product of hours worked and rate of pay;
print the data and the results;
```

With this algorithm, it should be possible, without too much trouble, to implement the corresponding program in almost any language since the fundamental constructs of most algorithmic

programming languages are similar. While we will discuss the features of C, similar features are usually available for most high level languages.

2.1.2 Translating the Algorithm to C

A program in a high level language, such as C, is called a **source program** or **source code**. (*Code* is a generic term used to refer to a program or part of a program in any language, high or low level). A program is made up of two types of items: *data* and *procedures*. *Data* is information we wish to process and is referred to using its *name*. *Procedures* are descriptions of the required steps to process the data and are also given names. In C, all procedures are called **functions**. A program may consist of one or more functions, but it must always include a function called *main*. This special function, `main()`, acts as a controller; directing all of the steps to be performed and is sometimes called the **driver**. The driver, like a conductor or a coordinator, may call upon other functions to carry out subtasks. When we refer to a function in the text, we will write its name followed by parentheses, e.g. `main()`, to indicate that this is the name of a function.

The program that implements the above algorithm in C is shown in Figure 2.1. Let us first look briefly at what the statements in the above program do during execution.

Any text between the markers, `/*` and `*/` is a **comment** or an explanation; it is not part of the program and is ignored by the compiler. However, comments are very useful for someone reading the program to understand what the program is doing. We suggest you get in the habit of including comments in your programs right from the first coding. The first few lines between `/*` and `*/` are thus ignored, and the actual program starts with the function name, `main()`. Parentheses are used in the code after the function name to list any information to be given to the function, called **arguments**. In this case, `main()` has no arguments. The *body* of the function `main()` is a number of *statements* between braces { and }, each terminated by a semi-colon.

The first two statements declare variables and their data types: `id_number` is an integer type, and `hours_worked`, `rate_of_pay`, and `pay` are floating point type. These statements indicate that memory should be allocated for these kinds of data and gives names to the allocated locations. The next statement writes or prints the title of the program on the screen.

The next three statements set the variables `id_number`, `hours_worked`, and `rate_of_pay` to some initial values: `id_number` is set to 123, `hours_worked` to 20.0, and `rate_of_pay` to 7.5. The next statement sets the variable `pay` to the product of the values of `hours_worked` and `rate_of_pay`. Finally, the last three statements print out the initial data values and the value of `pay`.

2.1.3 Running the Program

The program is entered and stored in the computer using an editor and saved in a *file* called `pay0.c`. The above *source program* must then be *compiled*, i.e. translated into a machine language *object program* using a *compiler*. Compilation is followed, usually automatically, by a *linking* process during which the compiled program is joined with other code for functions that may be defined

```
/*    File: pay0.c
      Programmer: Programmer Name
      Date: Current Date
      This program calculates the pay for one person, given the hours worked
      and rate of pay.
*/

main()
{     /* declarations */
      int id_number;
      float hours_worked,
            rate_of_pay,
            pay;

      /* print title */
      printf("***Pay Calculation***\n\n");

      /* initialize variables */
      id_number = 123;
      hours_worked = 20.0
      rate_of_pay = 7.5;

      /* calculate pay */
      pay = hours_worked * rate_of_pay;

      /* print data and results */
      printf("ID Number = %d\n", id_number);
      printf("Hours Worked = %f, Rate of Pay = %f\n",
                hours_worked, rate_of_pay);
      prinf("Pay = %f\n", pay);
}
```

Figure 2.1: Code for pay0.c

elsewhere. The C language provides a **library** of standard functions which are linked to every program and are available for use in the program. The end result is an *executable* machine language program also in a file. The executable machine language program is the only one that can be executed on a machine. We will use the term **compilation** to mean both compiling and linking to produce an executable program.

When the above program is compiled and executed on a computer, a sample session produces the following on the terminal:

```
***Pay Calculation***

ID Number = 123
Hours Worked = 20.000000, Rate of Pay = 7.500000
Pay = 150.000000
```

Throughout this text, we will show all information printed by the computer in `typewriter style characters`. As programs will frequently involve data entry by the user of the program during execution, in a sample session, all information typed in by the user will be shown in *slanted characters*.

2.2 Organization of C Programs — Simple Statements

We will now explain the syntax and semantics of the above program statements in more detail. Refer back to the source program in Figure 2.1 as we explain the statements in the program.

2.2.1 Comment Statements

As already mentioned, the text contained within `/*` and `*/` is called a comment. When the character pair `/*` is encountered, all subsequent text is ignored until the next `*/` is encountered. Comments are not part of the program; they are private notes that the programmer makes about the program to help one understand the logic. Comments may appear anywhere in a program but cannot contain other comments, i.e., they cannot be nested. For example:

```
/* This is a comment. /* Nested comments are not allowed */ this part
is not in a comment. */
```

The comment starts with the first `/*`. When the first matching `*/` is encountered after the word `allowed`, the comment is ended. The remaining text is not within the comment and the compiler tries to interpret the remaining text as program statement(s), most likely leading to errors.

2.2.2   Defining a Function — `main()`

To define a function in C, the programmer must specify two things: the **function header**, giving a name and other information about the function; and the **function body**, where the variables used in the function are defined and the statements which perform the steps of the function are specified.

The Function Header

In C, `main()` is the function that controls the execution of every program. The program starts executing with the first statement of `main()` and ends when `main()` ends. As we shall soon see, `main()` may call upon, i.e. use, other functions to perform subtasks.

The first line of any function is the function header which specifies the name of the function together with a parenthesized (possibly empty) argument list. In the above case, there is no argument list. We will discuss the concepts of arguments and argument lists in the next chapter.

The Function Body

The body of the function is contained within braces { and }. In C, a group of statements within braces is called a **block** which may contain zero or more statements and which may be nested, i.e. there may be blocks within blocks. A block is treated and executed as a single unit and is often called a **compound statement**. Such a compound statement may be used anywhere a statement can occur.

A program statement is like a sentence in English, except that it is terminated by a semi-colon. Statements within a block may be written in free form, i.e. words in programs may be separated by any amount of **white space**. (White space consists of spaces, tabs, or newlines (carriage returns)). Use of white space to separate statements and parts of a single statement makes programs more readable and therefore easier to understand.

The function body (as for any block) consists of two parts: *variable declarations* and a *list of statements*. Variable declarations will be described in more detail in the next section; however, all such declarations must occur at the beginning of the block. Once the first executable statement is encountered, no more declarations may occur for that block.

There are two types of statements used in our example (Figure 2.1); assignment statements and statements for printing information from the program. These will be discussed more below. The execution *control flow* proceeds sequentially in this program; when the function is executed, it begins with the first statement in the body and each statement is executed in succession. When the end of the block is reached, the function terminates. As we will soon see, certain control statements can alter this sequential control flow in well defined ways.

2.2.3 Variable Declarations

A **variable** is a language construct for identifying the data items used within a block. The declaration statements give names to these data items and specify the *type* of the item. The first two statements in our program are such declarations. The information we have in our task is the employee ID, the number of hours worked by the employee and the rate of pay. In addition, we will compute the total amount of pay for the employee and must declare a variable for this information. We have named variables for this information: `id_number`, `hours_worked`, `rate_of_pay`, and `pay`. We have also specified the type of each; for example, `id_number` is a whole number which requires an integer type, so the keyword `int` is used. The remaining data items are real numbers (they can have fractional values), so the keyword `float` is used to specify floating point type.

Variables of appropriate type (`int`, `float`, etc.) must be declared at the head of the block in which they are used. Several variables of the same type may be grouped together in a declaration, separated by commas.

```
int id_number;
float hours_worked,
      rate_of_pay,
      pay;
```

The names we have chosen for the variables are somewhat arbitrary; however, to make programs readable and easier to understand, variable names should be descriptive and have some meaning to the programmer. In programming languages, names are called **identifiers** and must satisfy certain rules.

First, identifiers may not be **keywords** (such as `int` and `float`) which have special meaning in C and are therefore reserved. All of these reserved words are listed in Appendix A. Otherwise, identifiers may include any sequence of lower and upper case letters, digits, and underscores; but the first character must be a letter or an underscore (though the use of an underscore as a first character is discouraged). Examples of legal identifiers include `PAD12`, `pad39`, `room_480`, etc. Alphabetic letters may be either lower case or upper case which are different; i.e. `PAY`, `Pay`, and `pay` are distinct identifiers for three different objects. There is no limit to the length of an identifier, however, there may be an implementation dependent limit to the number of significant characters that can be recognized by a compiler. (This means that if two identifiers do not differ in their first n characters, the compiler will not recognize them as distinct identifiers. A typical value for n might be 31).

The general form for a declaration statement is:

<type_specifier> <identifier>[, <identifier>...];

Throughout this text we will be presenting syntax specifications as shown above. The items surrounded by angle brackets (<>) are constructs of the language, for example <type_specifier> is a type specifier such as `int` or `float`, and <identifier> is a legal identifier. Items surrounded

main()

Size:	2	4	4	4
Type:	int	float	float	float
Name:	id_number	hours_worked	rate_of_pay	pay
Addr:	100	102	106	10A

Mem Cells

??	??	??	??
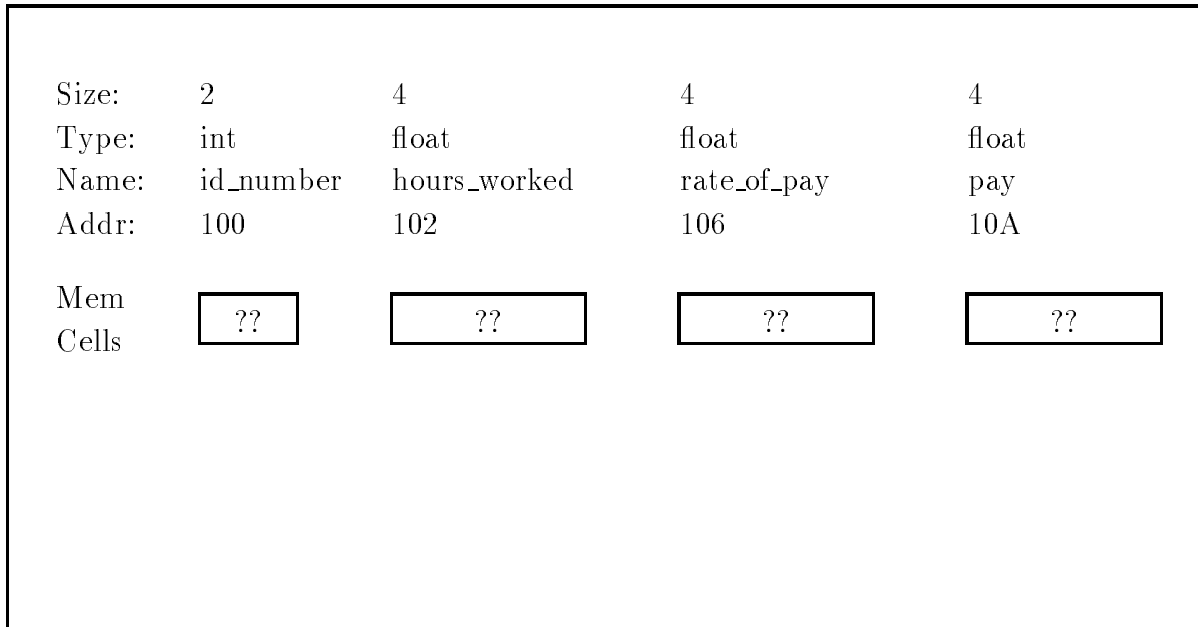
Figure 2.2: Allocation of Memory Cells or Objects

by square brackets ([ ]) are optional, i.e. they may or may not appear in a legal statement. The ellipsis (...) indicates one or more repetitions of the preceding item. Any other symbols are included in the statement exactly as typed. So, in words, the above syntax specification says that a declaration statement consists of a type specifier followed by an identifier and, optionally, one or more other identifiers separated by commas, all terminated by a semicolon.

As for the semantics (meaning) of this statement, a declaration statement does two things: allocates memory within the block for a data item of the indicated type, and assigns a name to the location. As we saw in Chapter 1, data is stored in the computer in a binary form, and different types of data require different amounts of memory. Allocating memory for a data item means to reserve the correct number of bytes in the memory for that type, i.e. choosing the address of the memory cells where the data item is to be stored.

Figure 2.2 shows memory allocation for the declarations in our program as it might occur on a 16 bit machine. The outer box shows that these variables have been allocated for the function **main()**. For each variable we show the size of the data item (in bytes), its type, name and assigned address assignment (in hex) above the box representing the cell itself. In the future, we will generally show only the memory cell and its name in similar diagrams. Note that the declaration statements **do not** put values in the allocated cells. We indicate this with the *??* in the boxes.

Memory cells allocated for specific data types are called objects. An object is identified by its starting address and its type. The type determines the size of the object in bytes and the encoding used to represent it. A variable is simply a named object which can be accessed by using its name. An analogy is gaining access to a house identified by the name of the person living there: Smith house, Anderson house, etc.

main()

Size:	2	4	4	4
Type:	int	float	float	float
Name:	id_number	hours_worked	rate_of_pay	pay
Addr:	100	102	106	10A
Mem Cells	123	20.0	7.5	??

Figure 2.3: Assignment of Values

Memory is automatically allocated for variables declared in a block when the block is entered during execution, and the memory is freed when the block is exited. Such variables are called **automatic variables**. The **scope** of automatic variables, i.e. the part of a program during which they can be used directly by name, is the block in which they are defined.

2.2.4 The Assignment Statement

The next three statements in our program assign initial values to variables, i.e. store initial values into objects represented by the variables. The assignment operator is =.

```
id_number = 123;
hours_worked = 20.0;
rate_of_pay = 7.5;
```

Each of the above statements stores the value of the expression on the right hand side of the assignment operator into the object referenced by the variable on the left hand side, e.g. the value stored in **id_number** is 123 (Figure 2.3). We will say the (current) value of **id_number** is 123. The value of a variable may change in the course of a program execution; for example, a new assignment can store new data into a variable. Storing new data overwrites the old data; otherwise, the value of a variable remains unchanged.

The "right hand side" of these three assignments is quite simple, a decimal constant. (The compiler will take care of converting the decimal number we use in the source code into its

main()

Size:	2	4	4	4
Type:	int	float	float	float
Name:	id_number	hours_worked	rate_of_pay	pay
Addr:	100	102	106	10A

Mem Cells:

123	20.0	7.5	150.0

Figure 2.4: Computation of `pay`

appropriate binary representation). However, in general the right hand side of an assignment may be an arbitrary *expression* consisting of constants, variable names and arithmetic operators (functions may also occur within expressions). For example, next, we calculate the product of the value of `hours_worked` and the value of `rate_of_pay`, and assign the result to the variable pay. The multiplication operator is `*`.

```
pay = hours_worked * rate_of_pay;
```

The semantics of the assignment operator is as follows: the expression on the right hand side of the assignment operator is first evaluated by replacing each instance of a variable by its current value and the operators are then applied to the resulting operands. Thus, the above right hand side expression is evaluated as:

```
20.0 * 7.5
```

The resulting value of the expression on the right hand side is then assigned to the variable on the left hand side of the assignment operator. Thus, the value of 20.0 * 7.5, i.e. 150.0, is stored in `pay` (Figure 2.4).

The above assignment expression may be paraphrased in English as follows:

   "SET pay TO THE VALUE OF hours_worked * rate_of_pay"

or,

"ASSIGN TO pay THE VALUE OF hours_worked * rate_of_pay"

The syntax of an assignment statement is:

<Lvalue>=<expression>;

The class of items allowed on the left hand side of an assignment operator is called an **Lvalue**, a mnemonic for *left value*. Of course, <Lvalue> must always reference an object where a value is to be stored. In what we've see so far, only a variable name can be an <Lvalue>. Later we will see other ways of referencing an object which can be used as an <Lvalue>.

As we can see from the above discussion, variables provide us a means for accessing information in our program. Using a variable on the left hand side of an assignment operator allows us to store a value in its memory cell. Variables appearing elsewhere in expressions cause the current value of the data item to be read and used in the expression.

In C every expression evaluated during execution results in a value. Assignment is also an expression, therefore also results in a value. Assignment expressions may be used anywhere expressions are allowed. The rule for evaluating an assignment expression is: evaluate the expression on the right hand side of the assignment operator, and assign the value to the variable on the left hand side. The value of the entire assignment expression is the value assigned to the left hand side variable. For example, x = 20 assigns 20 to x, and the value of the entire assignment expression is 20. So if we wrote y = x = 20, the variable y would be assigned the value of the expression x = 20, namely 20. In our programming example we have used assignment expressions as statements but ignored their values.

Any expression terminated by a semi-colon is a statement. Of course, a statement is typically written to perform some useful action. Some additional examples of expressions as statements are:

```
20;
5 + 10;
z = 20 * 5 + 10;
;
```

The last statement is an empty statement which does nothing. The expressions in the first two statements accomplish nothing since nothing is done with their values.

C has a rich set of operators for performing computations in expressions. The common arithmetic operators and their meanings are shown in Table 2.1. Two types of operators are shown; unary operators which take one operand, and binary operators which take two operands. The unary operators, + and - affect the sign of the operand. The binary operators are those you are familiar with, except possibly %. This is the **mod** operator, which we will describe below, but first one other point to make is that for the division operator /, if both operands are type integer, then integer division is performed, discarding and fractional part with the result also being type

Operator	Name	Example and Comments
+	plus sign	$+x$
−	minus sign	$-x$
+	addition	$x + y$
−	subtraction	$x - y$
*	multiplication	$x * y$
/	division	x/y
		if x, y are both integers,
		then x/y is integer,
		e.g., 5/3 is 1.
%	modulus	$x\%y$
		x and y MUST be integers:
		result is remainder of
		(x/y), e.g., 5%3 is 2.

Table 2.1: Arithmetic Operators

integer. Otherwise, a floating point result is produced for division. The mod operator evaluates to the remainder after integer division. Specifically, the following equality holds:

$$(x/y) * y + (x\%y) = x.$$

In words, if x and y are integers, multiplying the result of integer division by the denominator and adding the result of mod produces the numerator. We will see many more operators in future chapters.

2.2.5   Generating Output

Writing programs which declare variables and evaluate expressions would not be very useful if there were no way to communicate the results to the user. Generally this is done by printing (or writing) messages on the output.

Output of Messages

It is a good practice for a program to indicate its name or title when it is executed to identify the task which is being performed. The next statement in our program is:

```
printf("***Pay Calculation***\n\n");
```

The statement prints the program title on the terminal. This statement invokes the standard function `printf()` provided by every C compiler in a standard library of functions. The function `printf()` performs the subtask of writing information to the screen. When this statement is executed, the flow of control in the program passes to the code for `printf()`, and when `printf()` has completed whatever it has to do, control returns to this place in the program. These sequence of events is called a **function call**.

As can be seen in this case, a function can be called by simply using its name followed by a (possibly empty) pair of parentheses. Anything between the parentheses is called an **argument** and is information being sent to the function. In the above case, `printf()` has one argument, a string of characters surrounded by double quotes, called a **format string**. As we shall soon see, `printf()` can have more than one argument; however, the first argument of `printf()` must always be a format string. This `printf()` statement will write the following to the screen:

```
***Pay Calculation***
```

followed by two newlines. Note that all of the characters *inside* the double quotes have been printed (but not the quotes themselves), except those at the end of the string. The backslash character, '\', in the string indicates an **escape sequence**. It signals that the next character must be interpreted in a special way. In this case, '\n' prints out a newline character, i.e. all further printing is done on the next line of output. We will encounter other escape sequences in due time. Two newline escape sequences are used here; the first completes the line where "***Pay Calculation***" was written, and the second leaves a blank line in the output.

Output of Data

In addition to printing fixed messages, `printf()` can be used to print values of expressions by passing the values as additional arguments separated by commas. We print out values of the initial data and the result with the statements:

```
printf("ID Number = %d\n", id_number);
printf("Hours Worked = %f, Rate of Pay = %f\n",
          hours_worked, rate_of_pay);
printf("Pay = %f\n", pay);
```

The first argument of `printf()` must always be a format string and may be followed by any number of addition argument expressions (in this case simple variable names). As before, all regular characters in the format string are printed until the symbol %. The % and the following character, called a **conversion specification**, indicate that the value of the next argument is to be printed at this position in the output. The conversion character following % determines the format to be printed. The combination %d signals that a decimal integer value is to be printed at this position. Similarly, %f indicates that a decimal floating point value is to be printed at the

indicated position. (To write a % character itself, use %% in the format string). Each conversion specifier in the format string will print the value of one argument in succession.

The first `printf()` statement prints the value of `id_number` in the position where %d is located. The internal form of the value of `id_number` is converted to a decimal integer format and printed out. The output is:

    ID Number = 123

The next `printf()` writes the value of `hours_worked` at the position of the first %f, and the value of `rate_of_pay` at the position of the second %f. The internal forms are converted to decimal real numbers (i.e., floating point) and printed. The output is:

    Hours Worked = 20.000000, Rate of Pay = 7.500000

Observe that all regular characters in the format string, including the newline, are printed as before. Only the format conversion specification, indicated by a % followed by a conversion character d or f, is replaced by the value of the next unmatched argument. The floating point value is printed with six digits after the decimal point by default.

The final statement prints:

    pay = 150.000000

2.3   Testing the Program

As mentioned, the above program must be typed using an editor and saved in a file which we have called `pay0.c`. The program in C, a high level language, is called the **source program** or **source code**. It must be translated into the machine language for the particular computer being used. The machine language program is the only one that can be understood by the hardware.

A special program called a **compiler** is used to compile, i.e. translate a source program into a machine language program. The resulting machine language program is called the **object code** or **object program**. The object code may be automatically or optionally saved in a file. The terms **source file** and **object file** refer to the files containing the corresponding source code and object code.

The compiled object code is usually still not executable. The object code needs to be linked to machine language code for certain functions, e.g. code for library functions such as `printf()`, to create an executable machine language code file. A linker or a link editor is used for this step of linking disparate object codes. The linking step is usually automatic and transparent to the user. We will refer to the executable code variously as the **object code**, the **compiled program**, or the **load module**.

The executable code is then loaded into memory and run. The loading step is also transparent to the user; the user merely issues a command to run the executable code.

For many systems, the convention is that the source file name should end in .c as in pay0.c. Conventions for object file names differ; on some systems object files end in .obj, on others they end in .o, (Consult your system manuals for details). For compilation and execution, some systems require separate commands, one to compile a C program and the other to execute a compiled program. Other systems may provide a single command that both compiles and executes a program. Check your operating system and compiler manuals for details.

For Unix systems, the cc command, with many available options, is used for compilation. Examples are:

```
cc filename.c
cc -o outname filename.c
```

The first command line compiles the file filename.c and produces an executable file a.out. The second directs that the executable file is to be named outname. These programs are then run by typing the executable file name to the shell.

2.3.1 Debugging the Program

A program may have bugs, i.e. errors, in any of the above phases so these bugs must be removed; a process called **debugging**. Some bugs are easy to remove; others can be difficult. These bugs may appear at one of three times in testing the program: compile time, link time, and run time.

When a program is compiled, the compiler discovers syntax (grammar) errors, which occur when statements are written incorrectly. These compile time errors are easy to fix since the compiler usually pinpoints them reasonably well. The astute reader may have noticed there are bugs in the program shown in Figure 2.1. When the file pay0.c is compiled on a Unix C compiler, the following message is produced:

```
"pay0.c", line 21: syntax error at or near variable name "rate_of_pay"
```

This indicates some kind of syntax error was detected in the vicinity of line 21 near the variable name rate_of_pay. On examining the file, we notice that there is a missing semi-colon at the end of the previous statement:

```
hours_worked = 20.0
```

Inserting the semi-colon and compiling the program again eliminates the syntax error. In another type of error, the linker may not be able to find some of the functions used in the code so the linking process cannot be completed. If we now compile our file pay0.c again, we receive the following message:

```
/bin/ld: Unsatisfied symbols:
   prinf (code)
```

It indicates the linker was unable to find the function `prinf` which must have been used in our code. The linker states which functions are missing so link time errors are also easy to fix. This error is obvious, we didn't mean to use a function, `prinf()`, but merely misspelled `printf()` in the statement

```
    prinf("Pay = %f\n", pay);
```

Fixing this error and compiling the program again, we can successfully compile and link the program, yielding an executable file. As you gain experience, you will be able to arrive at a program free of compile time and link time errors in relatively few iterations of editing and compiling the program, maybe even one or two attempts.

A program that successfully compiles to an executable does not necessarily mean all bugs have been removed. Those remaining may be detected at run time; i.e. when the program is executed and may be of two types: computation errors and logic errors. An example of the former is an attempt to divide by zero. Once these are detected, they are relatively easy to fix. The more difficult errors to find and correct are program logic errors, i.e. a program does not perform its intended task correctly. Some logic errors are obvious immediately upon running the program; the results produced by the program are wrong so the statement that generates those results is suspect. Others may not be discovered for a long time especially in complex programs where logic errors may be hard to discover and fix. Often a complex program is accepted as correct if it works correctly for a set of well chosen data; however, it is very difficult to prove that such a program is correct in all possible situations. As a result, programmers take steps to try to avoid logic errors in their code. These techniques include, but are not limited to:

Careful Algorithm Development

As we have stated, and will continue to state throughout this text, careful design of of the algorithm is perhaps the most important step in programming. Developing and refining the algorithm using tools such as the structural diagram and flow chart discussed in Chapter 1 before any coding helps the programmer get a clear picture of the problem being solved and the method used for the solution. It also makes you think about what must be done before worrying about how to do it.

Modular Programming

Breaking a task into smaller pieces helps both at the algorithm design stage and at the debugging stage of program development. At the algorithm design stage, the modular approach allows the programmer to concentrate on the overall meaning of what operations are being done rather than the details of each operation. When each of the major steps are then broken down into smaller

steps, again the programmer can concentrate on one particular part of the algorithm at a time without worrying about how other steps will be done.

At debug time, this modular approach allows for quick and easy localization of errors. When the code is organized in the modules defined for the algorithm, when an error does occur, the programmer can think in terms of *what* the modules are doing (not *how*) to determine the most likely place where something is going wrong. Once a particular module is identified, the same refinement techniques can be used to further isolate the source of the trouble without considering all the other code in other modules.

Incremental Testing

Just as proper algorithm design and modular organization can speed up the debugging process, incremental implementation and testing can assist in program development. There are two approaches to this technique. The first is to develop the program from simpler instances of the task to more complex tasks as we are doing for the payroll problem in this chapter. The idea is to implement and test a simplified program and then add more complicated features until the full specification of the task is satisfied. Thus beginning from a version of the program known to be working correctly (or at least thoroughly tested), when new features are added and errors occur, the location of the errors can be localized to added code.

The second approach to incremental testing stems from the modular design of the code. Each module defined in the design can be implemented and tested independently so that there is high confidence that each module is performing correctly. Then when the modules are integrated together for the final program, when errors occur, again only the added code need be considered to find and correct them.

Program Tracing

Another useful technique for debugging programs begins after the program is coded, but before it is compiled and run, and is called a **program trace**. Here the operations in each statement of the program are verified by the programmer. In essence, the programmer is executing the program manually using pencil and paper to keep track changes to key variables. Diagrams of variable allocation such as those shown in Figures 2.2—2.4 may be used for this manual trace. Another way of manually tracing a program is shown in Figure 2.5. Here the changes in variables is seen associated with the statement which caused that change.

Program traces are also useful later in the debug phase. When an error is detected, a selective manual trace of a portion or module of a program can be very instrumental in pinpointing the problem. One word of caution about manual traces — care must be taken to update the variables in the trace according to the statement as written in the program, not according to the intention of the programmer as to what that statement should do.

Manual traces can become very complicated and tedious (one rarely traces an entire program),

```
/*    File: pay0.c
      Programmer: Programmer Name
      Date: Current Date
      This program calculates the pay for one person, given the
      hours worked and rate of pay.
*/

main()                                           PROGRAM TRACE
{                                                    hours_  rate_of_
      /* declarations */               id_number worked  pay          pay
      int id_number;                       ??
      float hours_worked,                          ??
            rate_of_pay,                                    ??
            pay;                                                     ??

      /* print title */
      printf("***Pay Calculation***\n\n");

      /* initialize variables */
      id_number = 123;                    123       ??      ??        ??
      hours_worked = 20;                  123      20.0     ??        ??
      rate_of_pay = 7.5;                  123      20.0    7.5        ??

      /* calculate results */
      pay = hours_worked * rate_of_pay;
                                          123      20.0    7.5      150.0

      /* print data and results */
      printf("ID Number = %d\n", id_number);
      printf("Hours Worked = %f, Rate of Pay = %f\n",
                hours_worked, rate_of_pay);
      printf("Pay = %f\n", pay);
}
```

Figure 2.5: Program Trace for pay0.c

however selective application of this technique is a valuable debugging tool. Later in this chapter we will discuss how the computer itself can assist us in generating traces of a program.

2.3.2   Documenting the Code

As a programmer, there are several "good" habits to develop for translating an algorithm into a source code program which support debugging as well as general understanding of the code. These habits fall under the topic of "coding for readability". We have already mentioned a few of these such as commenting the code and good choices of names for variables and functions. With good naming, the syntax of the C language allows for relatively good *self documenting code*; i.e. C source statements which can be read and understood with little effort.

Well documented code includes additional comments which clarify and amplify the meaning or intention of the statements. A good source for comments in your code are the steps of the algorithm you designed for the program. A well placed comment identifying which statements implement each step of the algorithm makes for easily understood programs.

Another good habit is to include judicious amounts of white space in your program. The C compiler would accept your program all written on one line; however, this would be very difficult for someone to read. Instead, space out your statements, separating groups of statements that perform logically different operations. It is also good to indent the statements in your program so that blocks are clearly identified at a glance. You will notice we have done that in Figure 2.1 and will continue throughout this text. There is no standard for indenting code, so you should choose a convention that is natural for you, as long as it is clear and you are consistent.

One last point: even though we have concentrated on the documentation of the code at the end of our discussion on this program, good documentation should be considered throughout the programming process. A bad habit to get into is to write the code and document it after it is working. A good habit is to include documentation in the code from the beginning.

In this section we have looked in detail at a C program that solves our simplified version of the payroll problem. The program in file `pay0.c` is not very useful since it can only be used to calculate pay for a specified set of data values because the data values are assigned to variables as constants in the program itself. If we needed to calculate the pay with some other employee, we would have to modify the program with new values and recompile and execute the program. For a program to be useful, it should be flexible enough to use any set of data values. In fact, the user should be able to enter a set of data during program execution, and the program should read and use these data values.

2.4   Input: Reading Data

To address the deficiency in our program mentioned above, the next task is to write a program that reads data typed by the user at the keyboard, calculates pay, and prints out the data and

the results. In this case, the program must communicate with the user to get the input data.

Task

PAY1: Same as PAY0, except that the data values `id_number`, `hours_worked`, and `rate_of_pay` should be read in from the keyboard.

The algorithm is the same as before except that the data is read rather than set:

```
print title of program;
read the data for id_number, hours_worked, and rate_of_pay;
set pay to the product of hours worked and rate of pay;
print the data and the results;
```

In the implementation of the above algorithm, we must read in data from the keyboard. In a C program, all communication with a user is performed by functions available in the standard library. We have already used `printf()` to write on the screen. Similarly, a function, `scanf()`, is available to read data in from the keyboard and store it in some object. `Printf()` performs the output function and `scanf()` performs the input function.

The function `scanf()` must perform several tasks: read data typed at the keyboard, convert the data to its internal form, and store it into an object. In C, there is no way for any function, including `scanf()`, to directly access a variable by its name defined in another function. Recall that we said the scope of a variable was the block in which it was defined, and it is only within this scope that a variable name is recognized. But if `scanf()` cannot directly access a variable in `main()`, it cannot assign a value to that variable. So how does `scanf()` store data into an object? A function can use the address of an object to indirectly access that object.

Therefore,`scanf()` must be supplied with the address of an object in which a data value is to be stored. In C, the *address of* operator, `&`, can be used to obtain the address of an object. For example, the expression `&x` evaluates to the address of the variable x. To read the id number from the keyboard and store the value into `id_number`, `hours_worked` and `rate_of_pay` we use the statements:

```
scanf("%d", &id_number);
scanf("%f", &hours_worked);
scanf("%f", &rate_of_pay);
```

The first argument of `scanf()` is a format string as it was for `printf()`. The conversion specification, %d, specifies that the input is in decimal integer form. `Scanf()` reads the input, converts it to an internal form, and stores it into an integer object whose address is given by the next unmatched argument. In this case, the value read is stored into the object whose address is `&id_number`, i.e. the value is stored into `id_number`. The remaining two scanf statements work similarly, except the conversion specification is `%f`, to indicate that a floating point number is to be read, converted

1	2	3		2	0	.	5	\n			

Figure 2.6: Keyboard Buffer

to internal form and stored in the objects whose addresses are &hours_worked and &rate_of_pay respectively. The type of the object must match the conversion specification, i.e. an integer value must be stored into an `int` type object and a floating point value into a `float` object.

To better understand how `scanf()` works, let us look in a little more detail. As a user types characters at the keyboard they are placed in a block of memory called a **buffer** (most but not all systems buffer their input). The function `scanf()` does not have access to this buffer until it is complete which is indicated when the user types the newline character, i.e. the RETURN key. (see Figure 2.6). The function `scanf()` then begins reading the characters in the buffer one at a time. When `scanf()` reads *numeric input*, it first skips over any leading white space and then reads a sequence of characters that make up a number of the specified type. For example, integers may only have a sign ($+ or -$) and the digits 0 to 9. A floating point number may possibly have a decimal point and the `e` or `E` exponent indicators. The function stops reading the input characters when it encounters a character that does not belong to the data type. For example, in Figure 2.6, the first `scanf()` stops reading when it sees the space character after the 3. The data is then converted to an internal form and stored into the object address specified in the argument. Any subsequent `scanf()` performed will begin reading where the last left off in the buffer, in this case at the space. When the newline character has been read, `scanf()` waits until the user types another buffer of data.

At this point we can modify our program by placing the `scanf()` statements in the code replacing the assignments to those variables. However, when we compile and execute the new program, nothing happens; no output is generated and the program just waits. The user does not know when a program is waiting for input unless the program prompts the user to type in the desired items. We use `printf()` statements to print a message to the screen telling the user what to do.

```
printf("Type ID Number: ");
scanf("%d", &id_number);
printf("Hours Worked: ");
scanf("%f", &hours_worked);
printf("Hourly Rate: ");
scanf("%f", &rate_of_pay);
```

The prompts are not necessary to read the data, without them, `scanf()` will read what is typed; but the user will not know when to enter the required data. We can now incorporate these statements into a program that implements the above algorithm shown as the file `pay1.c` in Figure 2.7. When the program is run, here is the sample output:

```
***Pay Calculation***
```

```
/*    File: pay1.c
      Programmer: Programmer Name
      Date: Current Date
      This program calculates the pay for one person with the
      hours worked and the rate of pay read in from the keyboard.
*/

main()
{
      /* declarations */
      int id_number;
      float hours_worked,
          rate_of_pay,
          pay;

      /* print title */
      printf("***Pay Calculation***\n\n");

      /* read data into variables */
      printf("Type ID Number: ");
      scanf("%d", &id_number);
      printf("Hours Worked: ");
      scanf("%f", &hours_worked);
      printf("Hourly Rate: ");
      scanf("%f", &rate_of_pay);

      /* calculate results */
      pay = hours_worked * rate_of_pay;

      /* print data and results */
      printf("\nID Number = %d\n", id_number);
      printf("Hours Worked = %f, Rate of Pay = %f\n",
              hours_worked, rate_of_pay);
      printf("Pay = %f\n", pay);
}
```

Figure 2.7: Code for pay1.c

```
Type ID Number:  123
Hours Worked:  20
Hourly Rate:  7.5

ID Number = 123
Hours Worked = 20.000000, Rate of Pay = 7.500000
Pay = 150.000000
```

Everything the user types at the keyboard is also echoed to the screen, and is shown here in *slanted characters*.

We have now seen two ways of storing data into objects: assignment to an object and reading into an object. Assignment stores the value of an expression into an object. Reading into an object involves reading data from the input, converting it to an internal form, and storing it in an object at a specified address.

The function `scanf()` can read several items of data at a time just as `printf()` can print several items of data at a time. For example,

```
scanf("%d %f %f", &id_number, &hours_worked, &rate_of_pay);
```

would read an integer and store it in `id_number`, read a float and store it in `hours_worked`, and read a float and store it in `rate_of_pay`. Of course, the prompt should tell the user to type the three items in the order expected by `scanf()`.

2.5   More C Statements

Our program `pay1.c` is still very simple. It calculates pay in only one way, the product of `hours_worked` and `rate_of_pay`. Our original problem statement in Chapter 1 called for computing overtime pay and for computing the pay for many employees. In this section we will look at additional features of the C language which will allow us to modify our program to meet the specification.

2.5.1   Making Decisions with Branches

Suppose there are different pay scales for regular and overtime work, so there are alternate ways of calculating pay: regular pay and overtime pay. Our next task is to write a program that calculates pay with work over 40 hours paid at 1.5 times the regular rate.

Task

PAY2: Same as PAY1, except that overtime pay is calculated at 1.5 times the normal rate.

For calculating pay in alternate ways, the program must make decisions during execution; so, we wish to incorporate the following steps in our algorithm:

```
if hours_worked is greater than 40.0,
     then calculate pay as the sum of
          excess hours at the overtime rate plus
          40.0 hours at regular rate;
     otherwise, calculate pay at the regular rate.
```

The program needs to make a decision: is `hours_worked` greater than 40.0? If so, execute one computation; otherwise, execute the alternate computation. Each alternate computation is implemented as a different *path* for program control flow to follow, called a **branch**. C provides a feature for implementing this algorithm form as follows:

```
if (hours_worked > 40.0)
     pay = 40.0 * rate_of_pay +
                 1.5 * rate_of_pay * (hours_worked - 40.0);
else
     pay = hours_worked * rate_of_pay;
```

The above `if` statement first evaluates the expression within parentheses:

```
hours_worked > 40.0
```

and if the expression is True, i.e. `hours_worked` is greater than 40.0, then the first statement is executed. Otherwise, if the expression is False, the statement following the `else` is executed. After one of the alternate statements is executed, the statement after the `if` statement will be executed. That is, in either case, the program control passes to the statement after the `if` statement.

The general syntax of an `if` statement is:

if (<expression>) <statement> [else <statement>]

The keyword `if` and the parentheses are required as shown. The two <statement>s shown are often called the **then clause** and the **else clause** respectively. The statements may be any valid C statement including a simple statement, a compound statement (a block), or even an empty statement. The else clause, the keyword `else` followed by a <statement>, is optional. Omitting this clause is equivalent to having an empty statement in the else clause. An `if` statement can be nested, i.e. either or both branches may also be `if` statements.
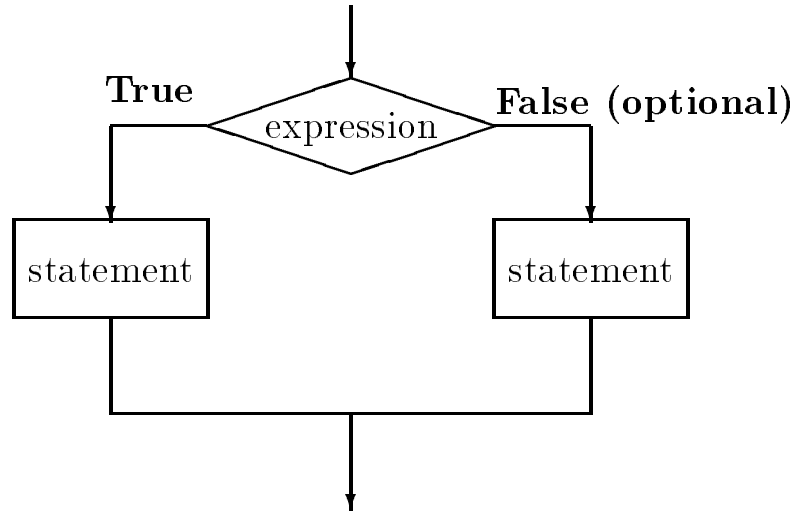
Figure 2.8: If statement control flow

The semantics of the `if` statement are that the expression (also called the **condition**) is evaluated, and the control flow branches to the then clause if the expression evaluates to True, and to the else clause (if any) otherwise. Control then continues with the statement immediately after the `if` statement. This control flow is shown in Figure 2.8.

It should be emphasized that only one of the two alternate branches is executed in an `if` statement. Suppose we wish to check if a number, x, is positive and also check if it is big, say greater than 100. Let us examine the following statement:

```
if (x > 0)
    printf("%d is a positive number\n", x);
else if (x > 100)
    printf("%d is a big number greater than 100\n", x);
```

If `x` is positive, say 200, the first `if` condition is True and the first `printf()` statement is executed. The control does not proceed to the `else` part at all, even though `x` is greater than 100. The else part is executed only if the first `if` condition is False. When two conditions overlap, one must carefully examine how the statement are constructed. Instead of the above, we should write:

```
if (x > 0)
    printf("%d is a positive number\n", x);
if (x > 100)
    printf("%d is a big number greater than 100\n", x);
```

Each of the above is a separate `if` statement. If `x` is positive, the first `printf()` is executed. In either case control then passes to the next `if` statement. If `x` is greater than 100, a message

is again printed. Another way of writing this, since (x > 100) is True only when (x > 0), we could write:

```
if (x > 0) {
        printf("%d is a positive number\n", x);
        if (x > 100)
                printf("%d is a big number greater than 100\n", x);
}
```

If (x > 0) is true, the compound statement is executed. It prints a message and executes the `if` (x > 100) ... statement. Suppose, we also wish to print a message when x is negative. We can add an else clause to the first `if` statement since positive and negative numbers do not overlap.

```
if (x > 0) {
        printf("%d is a positive number\n", x);
        if (x > 100)
                printf("%d is a big number greater than 100\n", x);
}
else if (x < 0)
        printf("%d is a negative number\n", x);
```

Something for you to think about: is there any condition for which no messages will be printed by the above code?

Returning to our payroll example, suppose we wish to keep track of both regular and overtime pay for each person. We can write the `if` statement:

```
if (hours_worked > 40.0) {
        regular_pay = 40.0 * rate_of_pay;
        overtime_pay = 1.5 * rate_of_pay * (hours_worked - 40.0);
}
else {
        regular_pay = hours_worked * rate_of_pay;
        overtime_pay = 0.0;
}
pay = regular_pay + overtime_pay;
```

Note: both clauses in this case are compound statements; each block representing a branch is treated as a single unit. Whichever branch is executed, that entire block is executed. If **hours_worked** exceeds 40.0, the first block is executed; otherwise, the next block is executed. Note, both blocks compute regular and overtime pay so that after the `if` statement the total pay can be calculated as the sum of regular and overtime pay. Also observe that we have used consistent data types in our expressions to forestall any unexpected problems. Since variables in the expressions are **float** type, we have used floating point constants 40.0, 1.5, and 0.0.

Operator	Meaning
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

Table 2.2: Relational Operators

Relational Operators

The greater than operator, >, used in the above expressions is called a **relational operator**. Other relational operators defined in C, together with their meanings are shown in Table 2.2 Note that for those relational operators having more than one symbol, the order of the symbols must be as specified in the table (>= not =>). Also take particular note that the equality relational operator is ==, NOT =, which is the assignment operator.

A relational operator compares the values of two expressions, one on each side of it. If the two values satisfy the relational operator, the overall expression evaluates to True; otherwise, it evaluates to False. In C, an expression that evaluates to False has the value of zero and an expression that evaluates to True has a non-zero value, typically 1. The reverse also holds; an expression that evaluates to zero is interpreted as False when it appears as a condition and expression that evaluates to non-zero is interpreted as True.

2.5.2 Simple Compiler Directives

In some of the improvements we have made so far to our program for PAY2, we have used numeric constants in the statements themselves. For example, in the code:

```
if (hours_worked > 40.0) {
    regular_pay = 40.0 * rate_of_pay;
    overtime_pay = 1.5 * rate_of_pay * (hours_worked - 40.0);
}
else {
    regular_pay = hours_worked * rate_of_pay;
    overtime_pay = 0.0;
}
pay = regular_pay + overtime_pay;
```

we use the constant 40.0 as the limit on the number of regular pay hours (hours beyond this are considered overtime), and the constant 1.5 as the overtime pay rate (time and a half). Use of

numeric constants (sometimes called "magic numbers") in program code is often considered bad style because the practice makes the program logic harder to understand and debug. In addition, the practice makes programs less flexible, since making a change in the values of numeric constants requires that the entire code be reviewed to find all instances where the "magic number" is used.

C, like many other programming languages, allows the use of symbolic names for constants in programs. This facility makes use of the C *preprocessor* and takes the form of **compiler directives**. Compiler directives are not, strictly speaking, part of the source code of a program, but rather are special directions given to the compiler about how to compile the program. The directive we will use here, the **define** directive, has syntax:

#define <symbol_name> <substitution_string>

All compiler directives, including **define**, require a **#** as the first non-white space character in a line. (Some older compilers require that **#** be in the first column of a line but most modern compilers allow leading white space on a line before **#**). The semantics of this directive is to define a string of characters, <substitution_string>, which is to be substituted for every occurrence of the symbolic name, <symbol_name>, in the code for the remainder of the source file. Keep in mind, a directive is not a statement in C, nor is it terminated by a semi-colon; it is simply additional information given to the compiler.

In our case, we might use the following compiler directives to give names to our numeric constants:

```
#define    REG_LIMIT       40.0
#define    OT_FACTOR       1.5
```

These directives define that wherever the string of characters REG_LIMIT occurs in the source file, it is to be replaced by the string of characters 40.0 and that the string OT_FACTOR is to be replaced by 1.5. With these definitions, it is possible for us to use REG_LIMIT and OT_FACTOR in the program statements instead of numeric constants. Thus our code would become:

```
if (hours_worked > REG_LIMIT) {
    regular_pay = REG_LIMIT * rate_of_pay;
    overtime_pay = OT_FACTOR * rate_of_pay * (hours_worked - REG_LIMIT);
}
else {
    regular_pay = hours_worked * rate_of_pay;
    overtime_pay = 0.0;
}
pay = regular_pay + overtime_pay;
```

The code is now more readable; it says in words exactly what we mean by these statements. Before compilation proper, the preprocessor replaces the symbolic constants with strings that constitute

actual constants; the string of characters `40.0` for the string `REG_LIMIT` and `1.5` for `OT_FACTOR` throughout the source program code.

The rules for the symbol names in directives are the same as those for identifiers. A common practice used by many programmers is to use upper case for the symbolic names in order to distinguish them from variable names. Remember, `define` directives result in a literal substitution without any data type checking, or evaluation. It is the responsibility of the programmer to use defines correctly. The source code is compiled after the preprocessor performs the substitutions.

The implementation of the PAY2 algorithm incorporating the above defines and other improvements discussed so far is shown in Figure 2.9. Note in the code, when the hours worked do not exceed `REG_LIMIT`, the overtime pay is set to zero. A constant zero value in a program code is not unreasonable when the logic is clear enough.

Here is a sample session from the resulting executable file:

```
***Pay Calculation***

Type ID Number:  456
Hours Worked:  50
Hourly Rate:  10

ID Number = 456
Hours Worked = 50.000000, Rate of Pay = 10.000000
Regular Pay = 400.000000, Overtime Pay = 150.000000
Total Pay = 550.000000
```

2.5.3 More on Expressions

Expressions used for computation or as conditions can become complex, and considerations must be made concerning how they will be evaluated. In this section we look at three of these considerations: precedence and associativity, the data type used in evaluating the expression, and logical operators.

Precedence and Associativity

Some of the assignment statements in the last section included expressions with more than one operator in them. The question can arise as to how such expressions are evaluated. Whenever there are several operators present in an expression, the order of evaluation depends on the **precedence** and **associativity** (or grouping) of operators as defined in the programming language. If operators have unequal precedence levels, then the operator with higher precedence is evaluated first. If operators have the same precedence level, then the order is determined by their associativity. The order of evaluation according to precedence and associativity may be overridden by using parentheses; expressions in parentheses are always evaluated first.

```
/*    File: pay2.c
      Programmer: Programmer Name
      Date: Current Date
      This program calculates the pay for one person, given the
      hours worked and rate of pay.
*/
#define   REG_LIMIT      40.0
#define   OT_FACTOR      1.5

main()
{     /* declarations */
      int id_number;
      float hours_worked,
            rate_of_pay,
            regular_pay, overtime_pay, total_pay;

      /* print title */
      printf("***Pay Calculation***\n\n");

      /* read data into variables */
      printf("Type ID Number: ");
      scanf("%d", &id_number);
      printf("Hours Worked: ");
      scanf("%f", &hours_worked);
      printf("Hourly Rate: ");
      scanf("%f", &rate_of_pay);

      /* calculate results */
      if (hours_worked > REG_LIMIT) {
          regular_pay = REG_LIMIT * rate_of_pay;
          overtime_pay = OT_FACTOR * rate_of_pay *
                                     (hours_worked - REG_LIMIT);
      }
      else {
          regular_pay = hours_worked * rate_of_pay;
          overtime_pay = 0.0;
      }
      total_pay = regular_pay + overtime_pay;
```

```
/* print data and results */
printf("\nID Number = %d\n", id_number);
printf("Hours Worked = %f, Rate of Pay = %f\n",
        hours_worked, rate_of_pay);
printf("Regular Pay = %f, Overtime Pay = %f\n",
        regular_pay, overtime_pay);
printf("Total Pay = %f\n", total_pay);
}
```

Figure 2.9: Code for pay2.c

Operator	Associativity	Type
+ , −	right to left	unary arithmetic
* , / , %	left to right	binary arithmetic
+ , −	left to right	binary arithmetic
< , <= , > , >=	left to right	binary relational
== , ! =	left to right	binary relational

Table 2.3: Precedence and Associativity of Operators

Table 2.3 shows the arithmetic and relational operators in precedence level groups separated by horizontal lines. The higher the group in the table, the higher its precedence level. For example, the precedence level of the binary operators *, /, and % is the same but it is higher than that of the binary operator group +, -. Therefore, the expression

```
x + y * z
```

is evaluated as

```
x + (y * z)
```

Associativity is also shown in the table. Left to right associativity means operators with the same precedence are applied in sequence from left to right. Binary operators are grouped from left to right, and unary from right to left. For example, the expression

```
x / y / z
```

is evaluated as

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

The precedence of the relational operators is lower than that of arithmetic operators, so if we had an expression like

```
x + y >= x - y
```

it would be evaluated as

```
(x + y) >= (x - y)
```

However, we will often include the parentheses in such expressions to make the program more readable.

From our payroll example, consider the assignment expression:

```
        overtime_pay = OT_FACTOR * rate_of_pay * (hours_worked - REG_LIMIT);
```

In this case, the parentheses are required because the product operator, *, has a higher precedence than the sum operator. If these parentheses were not there, the expression would be evaluated as:

```
        overtime_pay = (((OT_FACTOR * rate_of_pay) * hours_worked) - REG_LIMIT);
```

where what we intended was:

```
overtime_pay = ((OT_FACTOR * rate_of_pay) * (hours_worked - REG_LIMIT));
```

That is, the subtraction to be done first, followed by the product operators. There are several product operators in the expression; they are evaluated left to right in accordance with their associativity. Finally, the assignment operator, which has the lowest precedence, is evaluated.

Precise rules for evaluating expressions will be discussed further in Chapter 5 where a complete table of the precedence and associativity of all C operators will be given. Until then, we will point out any relevant rules as we need them and we will frequently use parentheses for clarity.

Data Types in Expressions

Another important consideration in using expressions is the type of the result. When operands of a binary operator are of the same type, the result is of that type. For example, a division operator applied to integer operands results in an integer value. If the operands are of mixed type, they are both converted to the type which has the greater range and the result is of that type; so, if the operands are `int` and `float`, then the result is floating point type. Thus, 8/5 is 1 and 8/5.0 is 1.6. The C language will automatically perform type conversions according to these rules; however, care must be taken to ensure the intent of the arithmetic operation is implemented. Let us look at an example.

Suppose we have a task to find the average for a collection of exam scores. We have already written the code which sums all the the scores into a variable `total_scores` and counted the number of exams in a variable `number_exams`. Since both of these data items are integer values, the variables are declared as type `int`. The average, however is a real number (has a fractional part) so we declared a variable `average` to be of type `float`. So we might write statements:

```
int total_scores, number_exams;
float  average;

...

average = total_scores / number_exams;
```

in our program. However, as we saw above, since `total_scores` and `number_exams` are both integers, the division will be done as integer division, discarding any fractional part. C will then automatically convert that result to a floating point number to be assigned to the variable `average`. For example, if `total_scores` is 125 and `number_exams` is 10, the the right hand side evaluates to the integer 12 (the fractional part is truncated) which is then converted to a `float`, 12.0 when it is assigned to `average`. The division has already truncated the fractional part, so our result will always have 0 for the fractional part of `average` which may be in error. We could represent either `total_scores` or `number_exams` as `float` type to force real division, but these quantities

Logical	C
AND	&&
OR	||
NOT	!

Table 2.4: Logical Operator Symbols in C

are more naturally integers. We would like to temporarily convert one or both of these values to a real number, only to perform the division. C provides such a facility, called the **cast operator**. In general, the syntax of the cast operator is:

(<type-specifier>) <expression>

which converts the value of <expression> to a type indicated by the <type-specifier>. Only the value of the expression is altered, not the type or representation of the variables used in the expression. The average is then computed as:

```
average = (float) total_scores / (float) number_exams;
```

The values of the variables are first both converted to `float` (e.g. 125.0 and 10.0), the division is performed yielding a `float` result (12.5) which is then assigned to `average`. We cast both variables to make the program more understandable. In general, it is good programming practice to cast variables in an expression to be all of the same type. After all, C will do the cast anyway, the cast is simply making the conversion clear in the code.

Logical Operators

It is frequently necessary to make decisions based on a logical combination of True and False values. For example, a company policy may not allow overtime pay for highly paid workers. Suppose only those workers, whose rate of pay is not higher than a maximum allowed value, are paid overtime. We need to write the pay calculation algorithm as follows:

```
if ((hours_worked > REG_LIMIT) AND (rate_of_pay <= MAXRATE))
     calculate regular and overtime pay
else
     calculate regular rate pay only, no overtime.
```

If hours worked exceeds the limit, `REG_LIMIT`, AND rate of pay does not exceed `MAXRATE`, then overtime pay is calculated; otherwise, pay is calculated at the regular rate. Such logical combinations of True and False values can be performed using **logical operators**. There are three generic logical operators: AND, OR, and NOT. Symbols used in C for these logical operators are

e1	e2	e1 && e2	e1 || e2	!e1
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Table 2.5: Truth Table for Logical Combinations

shown in Table 2.4 Table 2.5 shows logical combinations of True and False values and the resulting values for each of these logical operators. We have used T and F for True and False in the table. From the table we can see that the result of the AND operation is True only when the two expression operands are both True; the OR operation is True when either or both operands are True; and the NOT operation, a unary operator, is True when its operand is False.

We can use the above logical operators to write a pay calculation statement in C as follows:

```
if ((hours_worked > REG_LIMIT) && (rate_of_pay <= MAXRATE)) {
    regular_pay = REG_LIMIT * rate_of_pay;
    overtime_pay = OT_FACTOR * rate_of_pay *
                        (hours_worked - REG_LIMIT);
}
else {
    regular_pay = hours_worked * rate_of_pay;
    overtime_pay = 0;
}
```

(We assume that `MAXRATE` is defined using a define directive). We use parentheses to ensure the order in which expressions are evaluated. The expressions in the innermost parentheses are evaluated first, then the next outer parentheses are evaluated, and so on. If (hours_worked > REG_LIMIT) is True AND (rate_of_pay <= MAXRATE) is True, then the whole if expression is True and pay is calculated using the overtime rate. Otherwise, the expression is False and pay is calculated using regular rate.

In C, an expression is evaluated for True or False only as far as necessary to determine the result. For example, if (hours_worked > REG_LIMIT) is False, the rest of the logical AND expression need not be evaluated since whatever its value is, the AND expression will be False.

A logical OR applied to two expressions is True if either expression is True. For example, the above statement can be written in C with a logical OR operator, ||.

```
if ((hours_worked <= REG_LIMIT) || (rate_of_pay > MAXRATE)) {
    regular_pay = hours_worked * rate_of_pay;
    overtime_pay = 0;
}
```

```
    else {
        regular_pay = REG_LIMIT * rate_of_pay;
        overtime_pay = OT_FACTOR * rate_of_pay *
                            (hours_worked - REG_LIMIT);
    }
```

If either hours worked does not permit overtime OR the rate exceeds `MAXRATE` for overtime, calculate regular rate pay; otherwise, calculate regular and overtime pay. Again, if (`hours_worked <= REG_LIMIT`) is True, the logical OR expression is not evaluated further since the result is already known to be True. Precedence of logical AND and OR operators is lower than that of relational operators so the parentheses in the previous two code fragments are not required; however, we have used them for clarity.

Logical NOT applied to a True expression results in False, and vice versa. We can rewrite the above statement using a logical NOT operator, `!`, as follows:

```
    if ((hours_worked > REG_LIMIT) && !(rate_of_pay > MAXRATE)) {
        regular_pay = REG_LIMIT * rate_of_pay;
        overtime_pay = OT_FACTOR * rate_of_pay *
                            (hours_worked - REG_LIMIT);
    }
    else {
        regular_pay = hours_worked * rate_of_pay;
        overtime_pay = 0;
    }
```

If hours worked exceed `REG_LIMIT`, AND it is NOT True that rate of pay exceeds `MAXRATE`, then calculate overtime pay, etc. The NOT operator is unary and its precedence is higher than binary operators; therefore, the parentheses are required for the NOT expression shown.

2.5.4   A Simple Loop — `while`

Our latest program, `pay2.c`, still calculates pay for only one individual. If we have 50 people on the payroll, we must run the above program separately for each person. For our program to be useful and flexible, we should be able to repeat the same logical process of computation as many times as desired; i.e. it should be possible to write a program that calculates pay for any number of people.

Task

PAY3: Same as PAY2, except that the program reads data, computes pay, and prints the data and the pay for a known number of people.

Let us first see how to repeat the process of reading data, calculating pay, and printing the results a fixed number, say 10, times. To repeatedly execute an identical group of statements, we use what is called a **loop**. To count the number of times we repeat the computation, we use an integer variable, `count`. The logic we wish to implement is:

```
set count to 0
repeat the following as long as count is less than 10
    read data
    calculate pay
    print results
    increase count by 1
```

Initially, we set `count` to zero and we will repeat the process as long as `count` is less than 10. Each time we execute the loop, we increment `count` so that for each value of `count` (0, 1, 2, ..., 9), one set of data is processed. When `count` is 10, i.e. it is NOT less than 10, the repeating or looping is terminated.

The C language provides such a control construct; a `while` statement is used to repeat a statement or a block of statements. The syntax for a `while` statement is:

```
while ( <expression> ) <statement>
```

The keyword `while` and the parentheses are required as shown. The <expression> is a condition as it was for the `if` statement, and the <statement> may be any statement in C such as an empty statement, a simple statement, or a compound statement (including another `while` statement).

The semantics of the `while` statement is as follows. First, the while expression or condition, <expression>, is evaluated. If True, the <statement> is executed and the <expression> is evaluated again, etc. If at any time the <expression> evaluates to False, the loop is terminated and control passes to the statement after the `while` statement. This control flow for a `while` statement is shown in Figure 2.10.

To use the `while` statement to implement the algorithm above, there are several points to note about loops. The loop variable(s), i.e. variables used in the expression, must be initialized prior to the loop; otherwise, the loop expression is evaluated with unknown (garbage) value(s) for the variable(s). Second, if the loop expression is initially True, the loop variable(s) must be modified within the loop body so that the expression eventually becomes False. Otherwise, the loop will be an infinite loop, i.e. the loop repeats indefinitely. Therefore, a proper loop requires the following steps:

```
initialize loop variable(s)
while ( <expression> ) {
    . . .
    update loop variable(s)
}
```
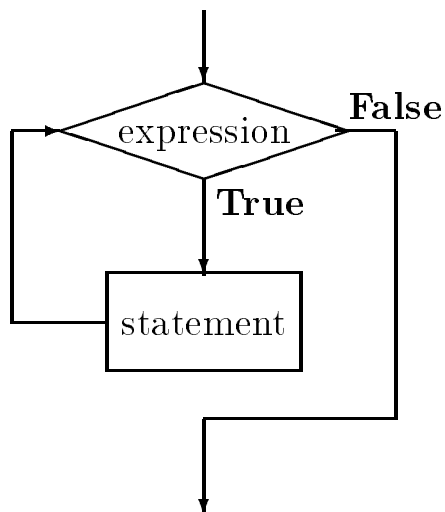
Figure 2.10: Control Flow for `while` statement

Keeping this syntax and semantics in mind, the code for the above algorithm fragment using a `while` loop is shown in Figure 2.11.

First, `count` is initialized to zero and tested for loop termination. The `while` statement will repeat as long as the while expression, i.e. (`count < 10`), is True. Since `count` is 0, the condition is true, so the body of the loop is executed. The loop body is a block which reads data, calculates pay, prints results, and increases the value of count by one. Except for updating `count`, the statements in the loop body are the same as those in the previous program in Figure 2.9. The count is updated by the assignment statement:

```
count = count + 1;
```

In this statement, the right hand side is evaluated first, i.e. one is added to the current value of `count`, then the new value is then stored back into `count`. Thus, the new value of count is one greater than its previous value. For the first iteration of the loop, `count` is incremented from 0 to 1 and the condition is tested again. Again (`count < 10`) is True, so the loop body is executed again. This process repeats until count becomes 10, (`count < 10`) is False, and the `while` statement is terminated. The program execution continues to the next statement, if any, after the `while` statement.

The above while loop is repeated ten times, once each for `count` = 0, 1, 2, ..., 9. We can also count the number of iterations to be performed as follows:

```
count = 10;
while (count > 0) {
    ...
    count = count - 1;
}
```

```
count = 0;
while (count < 10) {
    /* read data into variables */
    printf("Type ID Number: ");
    scanf("%d", &id_number);
    printf("Hours Worked: ");
    scanf("%f", &hours_worked);
    printf("Hourly Rate: ");
    scanf("%f", &rate_of_pay);

    /* calculate results */
    if (hours_worked > REG_LIMIT) {
        regular_pay = REG_LIMIT * rate_of_pay;
        overtime_pay = OT_FACTOR * rate_of_pay *
                                (hours_worked - REG_LIMIT);
    }
    else {
        regular_pay = hours_worked * rate_of_pay;
        overtime_pay = 0;
    }
    total_pay = regular_pay + overtime_pay;

    /* print data and results */
    printf("\nID Number = %d\n", id_number);
    printf("Hours Worked = %f, Rate of Pay = %f\n",
            hours_worked, rate_of_pay);
    printf("Regular Pay = %f, Overtime Pay = %f\n",
            regular_pay, overtime_pay);
    printf("Total Pay = %f\n", total_pay);

    /* update the count */
    count = count + 1;
}
```

Figure 2.11: Coding a While Loop

The initial value of count is 10 and the loop executes while (count > 0). Each time the loop is processed, the value of count is decremented by one. Eventually, count becomes 0, (count > 0) is False, and the loop terminates. Again, the loop is executed ten times for values of count = 10, 9, 8, ..., 1.

We can easily adapt the second approach to process a loop as many times as desired by the user. We merely ask the user to type in the number of people, and read into count. Here is the skeleton code.

```
printf("Number of people: ");
scanf("%d", &count);
while (count > 0) {
    ...
    count = count - 1;
}
```

We use the latter approach to implement the program for our task. The entire program for pay3.c is shown in Figure 2.12 A sample session from the execution of this program is shown below.

```
***Pay Calculation***

Number of people:  2

Type ID Number:  123
Hours Worked:  20
Hourly Rate:  7.5

ID Number = 123
Hours Worked = 20.000000, Rate of Pay = 7.500000
Regular Pay = 150.000000, Overtime Pay = 0.000000
Total Pay = 150.000000

Type ID Number:  456
Hours Worked:  50
Hourly Rate:  10

ID Number = 456
Hours Worked = 50.000000, Rate of Pay = 10.000000
Regular Pay = 400.000000, Overtime Pay = 150.000000
Total Pay = 550.000000
```

```
/*    File: pay3.c
      Programmer: Programmer Name
      Date: Current Date
      This program reads in hours worked and rate of pay and calculates
      the pay for a specified number of persons.
*/
#define   REG_LIMIT      40.0
#define   OT_FACTOR      1.5
main()
{
      /* declarations */
      int id_number, count;
      float hours_worked, rate_of_pay,
          regular_pay, overtime_pay, total_pay;



      /* print title */
      printf("***Pay Calculation***\n\n");

      printf("Number of people: ");
      scanf("%d", &count);
      while (count > 0) {
          /* read data into variables */
          printf("\nType ID Number: ");
          scanf("%d", &id_number);
          printf("Hours Worked: ");
          scanf("%f", &hours_worked);
          printf("Hourly Rate: ");
          scanf("%f", &rate_of_pay);

          /* calculate results */
          if (hours_worked > REG_LIMIT) {
              regular_pay = REG_LIMIT * rate_of_pay;
              overtime_pay = OT_FACTOR * rate_of_pay *
                                  (hours_worked - REG_LIMIT);
          }
          else {
              regular_pay = hours_worked * rate_of_pay;
              overtime_pay = 0.0;
          }
          total_pay = regular_pay + overtime_pay;
```

```
        /* print data and results */
        printf("\nID Number = %d\n", id_number);
        printf("Hours Worked = %f, Rate of Pay = %f\n",
                hours_worked, rate_of_pay);
        printf("Regular Pay = %f, Overtime Pay = %f\n",
                regular_pay, overtime_pay);
        printf("Total Pay = %f\n", total_pay);

        /* update the count */
        count = count - 1;
    }
}
```

Figure 2.12: Code for pay3.c

2.5.5 Controlling Loop Termination

The program in the last section illustrates one way to control how many times a loop is executed, namely counting the iterations. Rather than build the number of iterations into the program as a constant, pay3.c requires the user to type in the number of people for whom pay is to be computed. That technique may be sufficient sometimes, but the user may not be happy if each time a program is used, one has to count tens or hundreds of items. It might be more helpful to let the user signal the end of data input by typing a special value for the data. For example, the user can be asked to type a zero for the id number of the employee to signal the end of data (as long as zero is not an otherwise valid id number). This suggests another refinement to our task:

Task

PAY4: Same as PAY3, except that pay is to be calculated for any number of people. In addition, we wish to keep a count of the number of people, calculate the gross total of all pay disbursed, and compute the average pay. The end of data is signaled by a negative or a zero id number.

Logic for the while loop is quite simple. The loop repeats as long as id_number is greater than 0. This will also require us to initialize the id_number to some value before the loop starts and to update it within the loop body to ensure loop termination. For our task, we must also keep track of the number of people and the gross pay. After the while loop, we must calculate the average pay by dividing gross pay by the number of people. Here is the algorithm logic using the while loop construct.

```
    set gross pay and number of people to zero
    prompt user and read the first id number
    while (id number > 0) {
```

```
        read remaining data, compute pay, print data
        update number of people
        update gross pay
        prompt user and read next id number
    }
    set average pay to (gross pay / number of people)
```

Values of gross pay and number of people must be kept as cumulative values, i.e. each time pay for a new person is computed, the number of people must be increased by one, and gross pay must be increased by the pay for that person. Cumulative sum variables must be initialized to zero before the loop, similar to our counting variable in the last example; otherwise those variables will contain garbage values which will then be increased each time the loop is processed. Our algorithm is already "code like", and its implementation should be straightforward, but first let us consider the debugging process for the program.

As programs get more complex, manual program tracing becomes tedious; so let's let the program itself generate the trace for us. During program development, we can introduce `printf()` statements in the program to trace the values of key variables during program execution. If there are any bugs in program logic, the program trace will alert us. Such `printf()` statements facilitating the debug process are called **debug statements**. Once the program is debugged, the debug statements can be removed so that only relevant data is output. In our example, we will introduce debug statements to print values of gross pay and number of people.

In the program, we should not only prompt the user to type in an ID number but should also inform him/her that typing zero will terminate the data input. (Always assume that users do not know how to use a program). Prompts should be clear and helpful so a user can use a program without any special knowledge about the program. Figure 2.13 shows the program that implements the above algorithm.

Much of the code is similar to our previous program. We have introduced two additional variables, `number`, an integer counting the number of employees processed, and `gross`, a `float` to hold the cumulative sum of gross pay. Before the `while` loop, these variables are initialized to zero; otherwise only garbage values will be updated. Each time the loop body is executed, these values are updated: `number` by one, and `gross` by the new value of `total_pay`.

A debug statement in the `while` loop prints the updated values of `gross` and `number` each time the loop is executed. The output will begin with the word `debug` just to inform us that this is a debug line and will be removed in the final version of the program. Enough information should be given in debug lines to identify what is being printed. (A debug print out of line after line of only numbers isn't very useful for debugging). The values can alert us to possible bugs and to probable causes. For example, if we did not initialize `gross` to zero before the loop, the first iteration will print a garbage value for `gross`. It would instantly indicate to us that `gross` is probably not initialized to zero. We have also not indented the debug `printf()` statement to make it stand out in the source code.

Once the `while` loop terminates, the average pay must be computed as a ratio of `gross` and `number`. We have added another declaration at the beginning of the block for `average` and the

```c
/*    File: pay4.c
      Programmer: Programmer Name
      Date: Current Date
      This program reads in hours worked and rate of pay and calculates
      the pay for several persons. The program also computes the gross pay
      disbursed, number of people, and average pay. The end of data is
      signaled by a negative or a zero id number.
*/
#define    REG_LIMIT       40.0
#define    OT_FACTOR       1.5
main()
{
      /* declarations */
      int id_number, number;
      float hours_worked, rate_of_pay,
            regular_pay, overtime_pay, total_pay,
            gross, average;

      /* print title */
      printf("***Pay Calculation***\n\n");

      /* initialize cumulative sum variables */
      number = 0;
      gross = 0;
      /* initialize loop variables */
      printf("Type ID Number, 0 to quit: ");
      scanf("%d", &id_number);

      while (id_number > 0) {
            /* read data into variables */
            printf("Hours Worked: ");
            scanf("%f", &hours_worked);
            printf("Hourly Rate: ");
            scanf("%f", &rate_of_pay);

            /* calculate results */
            if (hours_worked > REG_LIMIT) {
                  regular_pay = REG_LIMIT * rate_of_pay;
                  overtime_pay = OT_FACTOR * rate_of_pay *
                                        (hours_worked - REG_LIMIT);
            }
            else {
                  regular_pay = hours_worked * rate_of_pay;
                  overtime_pay = 0;
            }
```

```
        total_pay = regular_pay + overtime_pay;

        /* print data and results */
        printf("\nID Number = %d\n", id_number);
        printf("Hours Worked = %f, Rate of Pay = $%f\n",
                    hours_worked, rate_of_pay);
        printf("Regular Pay = $%f, Overtime Pay = $%f\n",
                    regular_pay, overtime_pay);
        printf("Total Pay = $%f\n", total_pay);

        /* update cumulative sums */
        number = number + 1;
        gross = gross + total_pay;
        /* debug statements, print variable values */
printf("\ndebug: gross = %f, number = %d\n", gross, number);
        /* update loop variables */
        printf("\nType ID Number, 0 to quit: ");
        scanf("%d", &id_number);
    }
    if (number > 0) {
        average = gross / (float) number;
        printf("\n***Summary of Payroll***\n");
        printf("Number of people = %d, Gross Disbursements = $%f\n",
                    number, gross);
        printf("Average pay = $%f\n", average);
    }
}
```

Figure 2.13: Code for pay4.c

appropriate assignment statement to compute the average at the end. Note we have used the cast operator to cast **number** to a **float** for the division. This is not strictly necessary; the compiler will do this automatically; however, it is good practice to cast operands to like type in expressions so that we are aware of the conversion being done.

It is possible that no data was entered at all, i.e. the user enters 0 as the first id, in which case **number** is zero. If we try to divide **gross** by **number**, we will have a "divide by zero" run time error. Therefore, we check that **number** is greater than zero and only calculate the average and print the result when employee data has been entered.

With all of these changes made as shown in Figure 2.13, the program is compiled, and run resulting in the following sample session:

```
    ***Pay Calculation***
```

```
Type ID Number, 0 to quit:  123
Hours Worked:  20
Hourly Rate:  7.5


ID Number = 123
Hours Worked = 20.000000, Rate of Pay = $7.500000
Regular Pay = $150.000000, Overtime Pay = $0.000000
Total Pay = $150.000000


debug:  gross = 150.000000, number = 1


Type ID Number, 0 to quit:  456
Hours Worked:  50
Hourly Rate:  10


ID Number = 456
Hours Worked = 50.000000, Rate of Pay = $10.000000
Regular Pay = $400.000000, Overtime Pay = $150.000000
Total Pay = $550.000000


debug:  gross = 700.000000, number = 2


Type ID Number, 0 to quit:  0


***Summary of Payroll***
Number of people = 2, Gross Disbursements = $700.000000
Average pay = $350.000000
```

The debug lines show the changes in `gross` and `number` each time the loop is executed. The first such line shows the value of `gross` the same as that of the total pay and the value of `number` as 1. The next pass through the loop shows the variables are updated properly. The program appears to be working properly; nevertheless, it should be thoroughly tested with a variety of data input. Once the program is deemed satisfactory, the debug statements should be removed from the source code and the program recompiled.

2.5.6   More Complex Loop Constructs — Nested Loops

As we mentioned above, the <statement> that is the body of the loop can be any valid C statement and very often it is a compound statement. This includes a `while` statement, or a `while` statement with the block. Such a situation is called a **nested loop**. Nested loops frequently occur when several items in a sequence are to be tested for some property, and this testing itself requires repeated testing with several other items in sequence. To illustrate such a process, consider the following task:

Task

Find all prime numbers less than some maximum value.

The problem statement here is very simple; however, the algorithm may not be immediately obvious. We must first understand the problem.

A prime number is a natural number, i.e. 1, 2, 3, 4, etc., that is not exactly divisible by any other natural number, except 1 and itself. The number 1 is a prime by the above definition. The algorithm must find the other primes up to some maximum. One way to perform this task is to use a process called **generate and test**. In our algorithm, we will *generate* all positive integers in the range from 2 to a maximum (constant) value `PRIME_LIM`. Each generated integer becomes a candidate for a prime number and must be *test*ed to see if it is indeed prime. The test proceeds as follows: divide the candidate by every integer in sequence from 2 up to, but not including itself. If the candidate is not divisible by any of the integers, it is a prime number; otherwise it is not.

The above approach involves two phases: one generates candidates and the other tests each candidate for a particular property. The generate phase suggests a loop, each iteration of which performs the test phase, which is also a loop; thus we have a nested loop. Here is the algorithm.

```
set the candidate to 2
while (candidate < PRIME_LIM) {
     test the candidate for prime property
     print the result if a prime number
     generate the next candidate
}
```

In testing for the prime property, we will first assume that the candidate is prime. We will then divide the candidate by integers in sequence. If it is divisible by any of the integers excluding itself, then the candidate is not prime and we may generate the next candidate. Otherwise, we print the number as prime and generate the next candidate.

We need to keep track of the state of a candidate: it is prime or it is not prime. We can use a variable, let's call it `prime` which will hold one of two values indicating True or False Such a state variable is often called a **flag**. For each candidate, `prime` will be initially set to True. If the candidate is found to be divisible by one of the test integers, `prime` will be changed to False. When testing is terminated, if `prime` is still True, then the candidate is indeed a prime number and can be printed. This testing process can be written in the following algorithm:

```
set prime flag to True to assume candidate is a prime
set test divisor to 2
while (test divisor < candidate) {
     if remainder of (candidate/test divisor) == 0
          candidate is not prime
     else get the next test divisor in sequence
}
```

We will use the modulus (mod) operator, `%` described earlier, to determine the remainder of (`candidate / divisor`). Here is the code fragment for the above algorithm:

```
prime = TRUE;
divisor = 2;
while (divisor < candidate)  {
      if ((candidate % divisor) == 0)
           prime = FALSE;
      else
           divisor = divisor + 1;
}
```

where `TRUE` and `FALSE` are symbolic constants defined using the `define` compiler directive. The complete program is shown in Figure 2.14.

The program follows the algorithm step by step. We have defined symbols `TRUE` and `FALSE` to be 1 and 0, respectively. The final `if` statement uses the expression (`prime`) instead of (`prime == TRUE`); the result is the same. The expression (`prime`) is True (non-zero) if `prime` is `TRUE`, and False (zero) if `prime` is `FALSE`. Of course, we could have written the if expression as (`prime == TRUE`), but it is clear, and maybe more readable, as written.

We have included a debug statement in the inner loop to display the values of `candidate`, `divisor`, and `prime`. Once the we are satisfied that the program works correctly, the debug statement can be removed.

Here is a sample session with the debug statement and `PRIME_LIM` set to 8:

```
***Prime Numbers Less than 8***

1 is a prime number
2 is a prime number
debug: candidate = 3, divisor = 2 prime = 1
3 is a prime number
debug: candidate = 4, divisor = 2 prime = 1
debug: candidate = 4, divisor = 3 prime = 0
debug: candidate = 5, divisor = 2 prime = 1
debug: candidate = 5, divisor = 3 prime = 1
debug: candidate = 5, divisor = 4 prime = 1
5 is a prime number
debug: candidate = 6, divisor = 2 prime = 1
debug: candidate = 6, divisor = 3 prime = 0
debug: candidate = 6, divisor = 4 prime = 0
debug: candidate = 6, divisor = 5 prime = 0
debug: candidate = 7, divisor = 2 prime = 1
debug: candidate = 7, divisor = 3 prime = 1
debug: candidate = 7, divisor = 4 prime = 1
```

```
/*   File: prime.c
     Programmer: Programmer Name
     Date: Current Date
     This program finds all prime numbers less than PRIME_LIM.
*/
#define   PRIME_LIM       20
#define   TRUE            1
#define   FALSE           0

main()
{  int candidate, divisor, prime;

    printf("***Prime Numbers Less than %d***\n\n", PRIME_LIM);
    printf("%d is a prime number\n", 1);    /* print 1 */
    candidate = 2;                           /* start at candidate == 2 */

    while (candidate < PRIME_LIM) {    /* stop at candidate == 20 */
        prime = TRUE;                        /* for candidate, set prime to True */
        divisor = 2;                         /* initialize divisor to 2 */

        /* stop when divisor == candidate */
        while (divisor < candidate) {
printf("debug: candidate = %d, divisor = %d prime = %d\n",
        candidate, divisor,prime);

            /* if candidate is divisible by divisor, */
            /* candidate is not prime, set prime to False */
            if (candidate % divisor == 0)
                 prime = FALSE;
            divisor = divisor + 1;   /* update divisor */
        }
        if (prime)                              /* if prime is set to True, */
                                                /* print candidate. */
            printf("%d is a prime number\n", candidate);
        candidate = candidate + 1;        /* update candidate */
    }
}
```

Figure 2.14: Code for prime.c

```
debug: candidate = 7, divisor = 5 prime = 1
debug: candidate = 7, divisor = 6 prime = 1
7 is a prime number
```

We have shown part of a sample session with debug printing included. Notice, that the values printed for `prime` are 1 or 0; remember, `TRUE` and `FALSE` are symbolic names for 1 and 0 used in the source code program only. In this output the nested loops are shown to work correctly. For example, for candidate 5, divisor starts at 2 and progresses to 4; the loop terminates and the candidate is a prime number. A sample session without the debug statement is shown below.

```
***Prime Numbers Less than 20***

1 is a prime number
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
11 is a prime number
13 is a prime number
17 is a prime number
19 is a prime number
```

In looking at the debug output, you might see that the loop that tests for the prime property of a candidate is not an efficient one. For example, when `candidate` is 6, we know that it is not prime immediately after divisor 2 is tested. We could terminate the test loop as soon as `prime` becomes false (if it ever does). In addition, it turns out that a candidate needs to be tested for an even more limited range of divisors. The range of divisors need not exceed the square root of the candidate. (See Problem 6 at the end of the chapter).

2.6   Common Errors

In this section we list some common problems and programming errors that beginners often make. We also suggest steps to avoid these pitfalls.

1. Program logic is incorrect. This could be due to an incorrect understanding of the problem statement or improper algorithm design. To check what the program is doing, manually trace the program and use debug statements. Introduce enough debug statements to narrow down the code in which there is an error. Once an error is localized to a critical point in the code or perhaps to one or two statements, it is easier to find the error. Critical points in the code include before a loop starts, at the start of a loop, at the end of a loop and so forth.

2. Variables are used before they are initialized. This often results in garbage values occurring in the output of results. For example:

```
int x, y;

x = x * y;
```

There is no compiler error, x and y have unknown, garbage values. Be sure to initialize all variables.

3. The assignment operator, =, is used when an "equal to" operator, ==, is meant, e.g.:

```
while (x = y)
     ...

if (x = y)
     printf("x is equal to y\n");
```

There will be no compiler error since any valid expression is allowed as an if or while condition. The expression is True if non-zero is assigned, and False if zero is assigned. Always double check conditions to see that a correct equality operator, ==, is used.

4. Object names are passed, instead of addresses of objects, in function calls to scanf():

```
scanf("%d", n);            /* should be &n */
```

Again this is not a compile time error; the compiler will assume the value of n is the address of an integer object and will attempt to store a value in it. This often results in a run time addressing error. Make sure the passed arguments in scanf() calls are addresses of the objects where data is to be stored.

5. Loop variables are not initialized:

```
while (i < n)
     ...
```

i is garbage; the while expression is evaluated with unknown results.

6. Loop variables are not updated:

```
i = 0;
while (i < n) {
     ...
}
```

i is unchanged within the loop; it is always 0. The result is an infinite loop.

7. Loop conditions are in error. Suppose, a loop is to be executed ten times:

```
n = 10;
i = 0;
while (i <= n) {
    ...
    i = i + 1;
}
```

(i <= n) will be True for i = 0, 1, ..., 10, i.e. 11 times. The loop is executed one more time than required. Loop expressions should be examined for values of loop variables at the boundaries. Suppose n is zero; should the loop be executed? Suppose it is 1, suppose it is 10, etc.

8. User types in numbers incorrectly. This will be explained more fully in Chapter 4. Consider the loop:

```
while (x != 0) {
    ...
    scanf("%d", &x);
}
```

Suppose a user types: *23r*. An integer is read by scanf() until a non-digit is reached, in this case, until *r* is reached. The first integer read will be 23. However, the next time scanf() is executed it will be unable to read an integer since the first non-white space character is a non-digit. The loop will be an infinite loop.

9. Expressions should use consistent data types. If necessary, use a cast operator to convert one data type to another.

```
int sum, count;
float avg;

avg = sum / count;
```

Suppose sum is 30 and count is 7. The operation sum / count will be the integer value of 30 / 7, i.e. 4; the fractional part is truncated. The result 4 is assigned to a float variable avg as 4.0. If a floating point value is desired for the ratio of sum / count, then cast the integers to float:

```
avg = (float) sum / (float) count;
```

Now, the expression evaluates to 30.0 / 7.0 whose result is a floating point value 4.285 assigned to avg

2.7 Summary

In this chapter we have begun looking at the process of designing programs. We have stressed the importance of a correct understanding of the problem statement, and careful development of the algorithm to solve the problem. This is probably the most important, and sometimes the most difficult part of programming.

We have also begun introducing the *syntax* and *semantics* of the C language. We have seen how to define the special function, `main()` by specifying the *function header* followed by the *function body*, a collection of statements surrounded by brackets, { and }. The function body begins with variable declarations to allocate storage space and assign names to the locations, followed by the executable statements. Variable declarations take the form:

<type_specifier> <identifier>[, <identifier>...];

where <type_spec> may be either `int` or `float` for integers or floating point variables, respectively. (We will see other type specifiers in later chapters). We gave rules for valid <identifier>s used as variable names.

We have discussed several forms for executable statements in the language. The simplest statement is the assignment statement:

<Lvalue>=<expression>;

where (for now) <Lvalue> is a variable name and <expression> consists of constants, variable names and operators. We have presented some of the operators available for arithmetic computations and given rules for how expressions are evaluated. The assignment statement evaluates the expression on the right hand side of the operator `=` and stores the result in the object referenced by the <Lvalue>. We pointed out the importance of variable type in expressions and showed the cast operator for specifying type conversions within them.

(<type-specifier>) <expression>

We also described how the library function `printf()` can be used to generate output from the program, as well as how information may be read by the program at run time using the `scanf()` function.

We next discussed two program control constructs of the language: the `if` and `while` statements. The syntax for `if` statements is:

if (<expression>) <statement> [else <statement>]

where the <expression> is evaluated and if the result is True (non-zero) then the first <statement> (the "then" clause) is executed; otherwise, the <statement> after the keyword `else` (the "else" clause) is executed. For a `while` statement, the syntax is:

while (<expression>) <statement>

where the <expression> is evaluated, and as long as it evaluates to True, the <statement> is repeatedly executed.

In addition we discussed one of the simple compiler directives:

#define <symbol_name> <substitution_string>

which can be used to define symbolic names to character strings within the source code; used here for defining constants in the program.

With these basic tools of the language you should be able to begin developing your own programs to compile, debug and execute. Some suggestions are provided in the Problems Section below. In the next chapter, we will once again concentrate on the proper methods of designing programs, and in particular modular design with user defined functions.

2.8   Exercises

Given the following variables and their initializations:

```
int a, x, y, z;
float b, u, v, w;

x = 10; y= 20; z = 30;
u = 4.0; v = 10.0;
```

What are the values of the expressions in each of the following problems:

1. (a)   a = x - y - z;
   (b)   a = x + y * z;
   (c)   a = z / y + y;
   (d)   a = x / y / z;
   (e)   a = x % y % z

2. (a)   a = (int) (u / v);
   (b)   a = (int) (v / u);
   (c)   b = v - u;
   (d)   b = v / u / w;

3. What are the results of the following mod operations:

```
(a)    5  %   3
(b)   -5  %   3
(c)    5  % -3
(d)   -5  % -3
\item
\begin{verbatim}
(a)   (x <= y && x >= z)
(b)   (x <= y || x >= z)
(c)   (x <= y && !(x >= z))
(d)   (x = y && z > y)
(e)   (x == y && z > y)
```

4. Under what conditions are the following expressions True?

```
(a)   (x = y && y = z)
(b)   (x == y && y == z)
(c)   (x == y || y == z)
(d)   (x >= y && x <= z)
(e)   (x > y && x < z)
```

5. Make required corrections in the following code.

   (a)
```
main()
{    int n;

     scanf("%d", n);
}
```

   (b)
```
main()
{    float n;

     printf("%d", n);
}
```

   (c)

```
main()
{    int n1, n2;

     if (n1 = n2)
          printf("Equal\n");
     else
          printf("Not equal\n");
}
```

6. Find and correct errors in the following program that is supposed to read ten numbers and print them.

```
main()
{    int n, count;

     scanf("%d", &n);
     while (count < 10) {
          printf("%d\n", n);
          scanf("%d", &n);
     }
}
```

7. We wish to print integers from 1 through 10. Check if the following loop will do so correctly.

```
i = 1;
while (i < 10) {
     printf("%d\n", i);
     i =  i + 1;
}
```

8. Suppose a library fine for late books is: 10 cents for the first day, 15 cents per day thereafter. Assume that the number of late days is assigned to a variable `late_days`. Check if the following will compute the fine correctly.

```
if (late_days == 1)
        fine = 0.10;
else
        fine = late_days * 0.15;
```

2.9   Problems

1. Write a program that reads three variables x, y, and z. The program should check if all three are equal, or if two of the three are equal, or if none are equal. Print the result of the tests. Show the program with manual trace.

2. Velocity of an object traveling at a constant speed can be expressed in terms of distance traveled in a given time, If distance, s, is in feet and time, t, is in seconds, the velocity in feet per second is:

$$v = d/t$$

   Write a program to read distance traveled and time taken, and calculate the velocity for a variety of input values until distance traveled is zero. Print the results for each case. Show a manual trace.

3. Acceleration of an object due to gravity, g, is 32 feet per second per second. The velocity of a falling body starting from rest at time, t, is given by:

$$v = g * t$$

   The distance traveled in time, t, by a falling body starting from rest is given by:

$$d = g * t * t/2$$

   Write a program that repeatedly reads experimental values of time taken by a body to hit the ground from various heights. The program calculates for each case: the height of the body and the velocity of the body when it hits the ground.

4. Write a program that reads a set of integers until a zero is entered. Excluding zero, the program should print a count of and a sum of:

   (a) positive numbers

   (b) negative numbers

   (c) even numbers

   (d) odd numbers

   (e) positive even numbers

   (f) negative odd numbers.

   (g) all numbers

   Use debug statements to show cumulative sums as each new number is read and processed.

5. We wish to convert miles to kilometers, and vice versa. Use the loose definition that a kilometer is 5.0 / 8.0 of a mile. Write a program that generates two tables: a table for kilometer equivalents to miles for miles 1 through 10, and a table for mile equivalents of kilometers for kilometers from 1 to 20.

6. Improve the program `prime.c` of Section 2.5.6 in the following ways:

(a) Terminate the inner loop as soon as it is detected that the number is not prime.

(b) Test each candidate only while (`divisor * divisor <= candidate`).

(c) Test only candidates that are odd numbers greater than 3.

For each of these improvements, how many times is the inner loop executed when `PRIME_LIM` is 20? How does that compare to our original program?

7. Write a program to generate Fibonacci numbers less than 100. Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, etc. The first two Fibonacci numbers are 1 and 1. All other numbers follow the pattern: a Fibonacci number is the sum of previous two Fibonacci numbers in the sequence. In words, the algorithm for this problem is as follows:

We will use two variables, `prev1` and `prev2`, such that `prev1` is the last fibonacci number and `prev2` is the one before the last. Print the first two fibonacci numbers, 1 and 1; and initialize `prev1` and `prev2` as 1 and 1. The new `fib_number` is the sum of the two previous numbers, `prev1` and `prev2`; the new `fib_number` is now the last fibonacci number and `prev1` is the one before the last. So, save `prev1` in `prev2` and save `fib_number` in `prev1`. Repeat the process while `fib_number` is less than 100.

8. (Optional) Write a program to determine the largest positive integer that can be stored in an `int` type variable. An algorithm to do this is as follows:

Initialize a variable to 1. Multiply by 2 and add 1 to the variable repeatedly until a negative value appears in the variable. The value of the variable just before it turned negative is the largest positive value.

The above follows from the fact that multiplying by 2 shifts the binary form to the left by one position. Adding one to the result makes all ones in the less significant part and all zeros in the more significant part. Eventually a 1 appears in the leading sign bit, i.e. a negative number appears. The result just before that happens is the one with all ones except for the sign bit which is 0. This is the largest positive value.

9. (Optional) Write a program to determine the negative number with the largest absolute value.

10. Write a program that reads data for a number of students and computes and prints their GPR. For each student, an id number and transcript data for a number of courses is read. Transcript data for each course consists of a course number (range 100-900), number of credits (range 1-6), and grade (range 0-4). The GPR is the ratio of number of total grade points for all courses and the total number of credits for all courses. The number of grade points for one course is the product of grade and credits for the course. The end of transcript data is signaled by a zero for the course number; the end of student data is signaled by a zero id number.

Chapter 3

Designing Programs Top Down

As program tasks become more complex, it is easier to think about the problem and design the algorithm for the task at hand by breaking the complex task into smaller and simpler *subtasks* and then solve each of the subtasks independently. We do this all the time in everyday life, for example, suppose you need milk for your kid's dinner. A complete algorithm for solving this problem might begin:

```
find the car keys
go to the garage
get in the car
put the key in the ignition
start the car
back the car out of the driveway
...
```

However; when we are worried about feeding the kids, we do not plan our algorithm in such detail. Instead our algorithm might be:

```
drive to the store
buy milk
drive home
```

where each of the steps in this algorithm is a subtask that may involve many steps itself.

We can do the same kind of *modular design* for our programming tasks: begin by thinking at a more abstract level about the major steps to be done, and then for each of these subtasks, design a separate algorithm to solve it. Each program subtask may then be implemented either by a set of statements or by a separate *function*. The advantages of a function are that it hides details of the actual computations from the main body of the code, and it can even be called upon to perform a subtask repeatedly by one or more other functions. In particular, well designed functions can
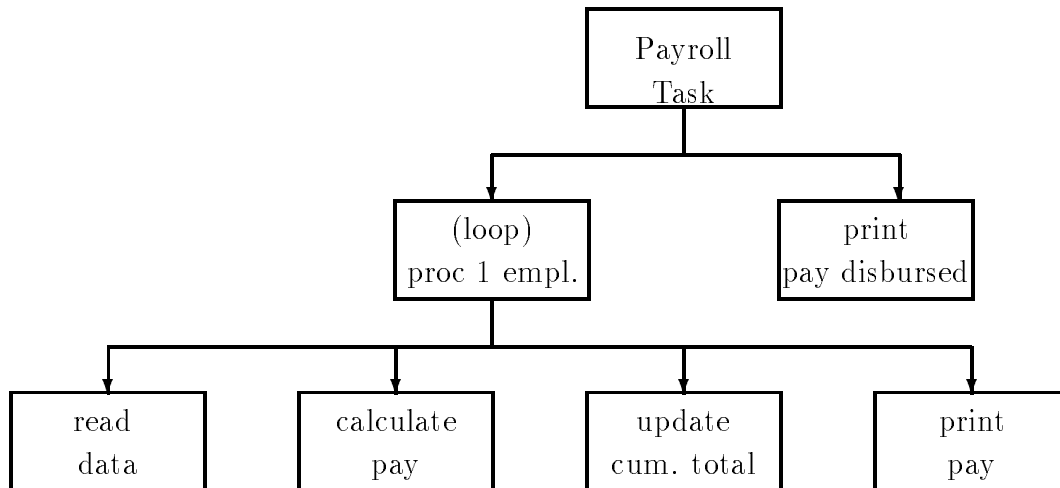
Figure 3.1: Structural Diagram for Payroll Task

be used in a variety of programs. (An example from the above might be driving; it is the same operation in the first and last steps of our algorithm; only the start and destination are different).

   In this chapter we will discuss this method of modular design of algorithms and the programs that implement them. We will see how functions may be used in a C program, and how new functions may be defined in the program. As usual, we will look at both the syntax and semantics of this programming construct. Next we will look in more detail at the *macro* facilities provided by the C preprocessor (briefly discussed in Chapter 2) and how these can be used to make programs more readable. The we describe how your programs can interact with the Operating System to perform I/O. Finally we continue our discussion of guidelines for debugging and common errors.

3.1   Designing the Algorithm with Functions

As mentioned above, for complex problems our goal is to divide the task into smaller and simpler tasks during algorithm design. We have seen this technique already in Chapter 1 in our use of a *structural diagram* while developing the algorithm. Figure 3.1 repeats the structural diagram for our payroll task. Here we have divided the payroll task at first into 2 subtasks: processing employees one at a time in a loop, and printing the results. The "processing one employee" subtask is then further divided into four steps: reading data, calculating pay, updating the cumulative total, and printing the pay. In the final implementation of our algorithm, `pay4.c`, we implemented each step using a sequence of statements. The resulting code grew to be rather large, especially for the "calculate pay" step where we had to consider details such as overtime and regular pay. Such details are not important to our understanding of the overall *logic* of the program. However it is to be done, all that we want to do in that step is calculate the pay for one employee as is simply and clearly stated in the algorithm. Calculating pay is an ideal candidate for being implemented as a function.

We will show how to do this shortly, but first it should be pointed out that we have already been using functions to hide the details of tasks in the code we have written. For both the "read data" and "print pay" blocks in the diagram (and the corresponding steps in the algorithm) we have used the built-in library functions, `scanf()` and `printf()`. Many operations are involved in reading the user's typed in data, converting it to its internal representation, and storing it in a variable; however all of this processing is *hidden* by the function `scanf()`. At this point, we do not need to know (and maybe don't care) how it is done, just that it is done correctly.

The important thing here is that top level program logic can use functions without regard to their details. At the next lower level, each function used in the top level program logic can be written in terms of yet lower level functions, and so on. The goal is to arrive at subtasks that are simple to implement with relatively few statements. This approach is called the **top down approach** or **modular programming**. A top down approach is an excellent aid to program development. If the subtasks are simple enough, it also helps produce bug-free reliable programs.

3.1.1 Implementing the Program with Functions

Abstractly, a function can be viewed a a piece of code which, when given sufficient information, performs some subtask and returns the result, a value. Returning to our example, if a function, `calc_pay()`, is used to calculate pay, it will need enough information to perform the computation. In this case the data it needs is the number of hours worked and the rate of pay. As we have stated before, variables, such as `hours_worked` and `rate_of_pay`, defined in a block are only *known*, i.e. can be accessed, within that block. So we cannot give `calc_pay()` direct access to variables defined in other functions, in this case `main()`. However, `calc_pay()` does not need direct access to the variables, it only needs the values to be used for the computation. So we can give a function the values it needs by passing them as **arguments**. We can do this by writing an expression, called a **function call**, giving the name of the function and expressions for the values of the arguments, e.g.:

```
calc_pay(hours_worked, rate_of_pay)
```

The arguments passed are the *values* of `hours_worked` as the first argument, and `rate_of_pay` as the second argument. Given this data we know (or at this point simply believe) that the function does the right thing and returns with a value, the total pay. We say that the function call *evaluates to a value* just as any other expression. The function `calc_pay()` can now be used in `main()` as follows:

```
total_pay = calc_pay(hours_worked, rate_of_pay);
```

In summary, the function `main()` calls `calc_pay()` to perform a task using a set of values. The values are passed as a parenthesized list of data items (which can be any valid expressions) separated by commas. The expressions that appear in such a statement calling the function are called **arguments**. The values of these arguments are received by the called function, `calc_pay()`, which

uses them to perform the desired subtask. Finally, `calc_pay()` returns the value of total pay to the calling function, `main()`, where it is assigned to the variable, `total_pay`.

The value returned by `calc_pay()` will be the total pay calculated using the values of arguments passed to it. Here are a few additional examples of function calls used in an assignment expression:

```
total_pay = calc_pay(30.0, 10.0);  /* calc_pay() returns 300.0, */
                                    /* which is stored in total_pay. */
total_pay = calc_pay(20.0, 10.0);  /* total_pay is assigned 200.0. */
```

A function call is an expression and has a value. Just as we had to declare the data types of variables to the compiler, we must also declare the data type of a function. This declaration also includes the number of arguments the function requires and their types. For example, here is a declaration for `calc_pay()`:

```
float calc_pay(float hours, float rate);
```

The declaration states that `calc_pay()` is a function because the identifier `calc_pay` is followed by a parenthesized list of arguments, that it requires two `float` arguments, and that it is of `float` type, i.e. it returns a `float` value. This declaration statement for a function (notice it is terminated by a semi-colon) is called a **prototype statement** because it gives the *prototype* (or the form) for calls to the function. In general, we will refer to the list of data expected to be passed to a function as specified in the prototype statement as a **parameter list** and an individual data item in this list as a **parameter**. (Sometimes, however, the terms parameter and argument are used interchangeably). The names of the parameters in a prototype statment are optional; but including well chosen names for parameters can make the declaration more meaningful. These parameter names are dummy names which have no relation to the names of arguments in a function call or parameters in the function definition (described in the next section).

Let us implement the top level program logic using the function `calc_pay()` to calculate pay. The code is shown in Figure 3.2 and for simplicity, we have not included calculation of `gross` and `average_pay`.

Figure 3.3 shows the behavior of the function call pictorially. The box labeled **main()** represents the function `main()` in our program and contains memory cells for variables declared in `main()` labeled with their names (e.g. `hours_worked`). The box labeled **calc_pay()** represents the function `calc_pay()`. At this point we do not know anything about the internals of this box such as what variables are declared, and what statements will be executed; but at this point we do not need to know this information. The box shows all of the information we need to know; namely that the function expects two `float` type arguments to be passed and will return a `float` type result. The dashed lines in the figure show that, for the call we have written in `main()`:

```
total_pay = calc_pay(hours_worked, rate_of_pay);
```

```
/*    File: pay5.c
      Programmer: Programmer Name
      Date: Current Date
      The program gets payroll data, calculates pay, and prints out
      the results for a number of people. A separate function is used
      to calculate total pay.
*/
#define   REG_LIMIT     40.0
#define   OT_FACTOR     1.5

main()
{
    /* declarations */
    int id_number;
    float hours_worked, rate_of_pay, total_pay;
    float calc_pay(float hours, float rate);

    /* print title */
    printf("***Pay Calculation***\n");

    /* initialize loop variables */
    printf("\nType ID Number, zero to quit: ");
    scanf("%d", &id_number);

    while (id_number > 0) {
        /* read data into variables */
        printf("Hours Worked: ");
        scanf("%f", &hours_worked);
        printf("Hourly Rate: ");
        scanf("%f", &rate_of_pay);

        /* calculate pay */
        total_pay = calc_pay(hours_worked, rate_of_pay);

        /* print data and results */
        printf("\nID Number = %d\n", id_number);
        printf("Hours Worked = %f, Rate of Pay = $%6.2f\n",
                  hours_worked, rate_of_pay);
        printf("Total Pay = $%10.2f\n", total_pay);

        /* update loop variables */
        printf("\nType ID Number, zero to quit: ");
        scanf("%d", &id_number);
    }
}
```

Figure 3.2: Code for pay5.c driver
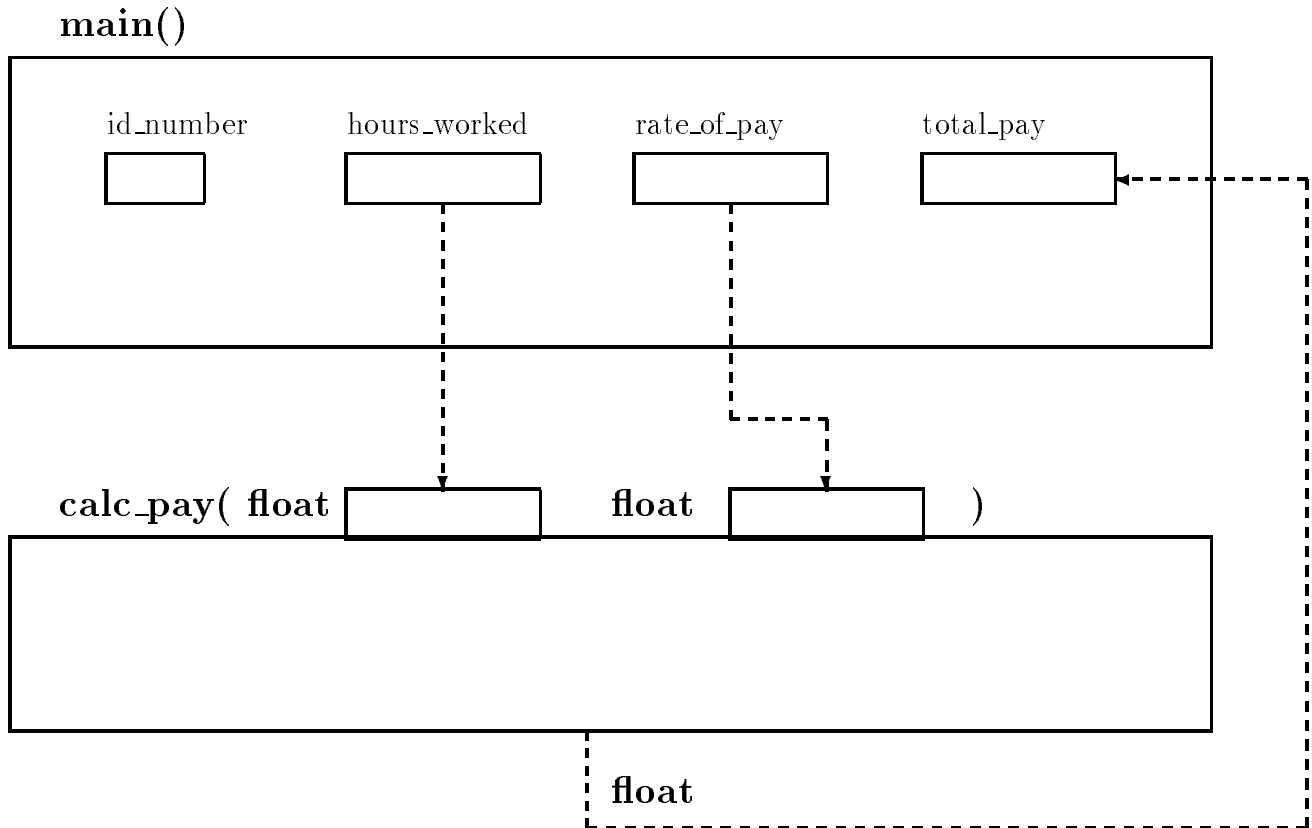
Figure 3.3: Function Call to `calc_pay()`

the first argument, the value of `hours_worked`, is passed to the first parameter of `calc_pay()`, and the second argument, the value of `rate_of_pay`, is passed to the second parameter. The return value from `calc_pay()` is placed in the variable `total_pay` by `main()`.

In summary, the function `main()` represents the overall logic of the program. The details of how pay is actually computed does not change the overall logic. Of course, the program in Figure 3.2 is not yet complete since we have not written the function `calc_pay()`. If an attempt is made to compile the program at this point, there will be a linker error message stating that the function `calc_pay()` cannot be found. Only when the function is written is the program complete and may be compiled and executed.

3.2   Defining Functions

A function is defined by writing the source code for it. Just as for `main()`, defining the function consists of giving a *function header* and a *function body*. The code for `calc_pay()` is shown in Figure 3.4. (It is included in the same source file as the code in Figure 3.2). Let us look at the function header first.

```
/* File: pay5.c - continued */
/* Function calculates and returns total pay */
float calc_pay(float hours, float rate)
{    float regular, overtime, total;

printf("\ndebug:entering calc_pay(): hours = %f, rate = %f\n",
                hours, rate);

    if (hours > REG_LIMIT) {
        regular = REG_LIMIT * rate;
        overtime = OT_FACTOR * rate * (hours - REG_LIMIT);
    }
    else {
        regular = hours * rate;
        overtime = 0;
    }
    total = regular + overtime;
printf("debug:returning from calc_pay(): %f\n", total);
    return total;
}
```

Figure 3.4: Code for `calc_pay()`

```
float calc_pay(float hours, float rate)
```

The header specifies that the name of the function is `calc_pay`, and that the function returns a `float` value. It also lists the parameters and their types, in this case there are two formal parameters, `hours` and `rate`, each of type `float`. Notice that the function header is very similar to the prototype statement for the function, with two notable exceptions. First, there is no semi-colon at the end, indicating that this is the definition of the function, not a declaration. Second, in the function header, the variable names in the parameter list are required, and this list is sometimes called the **formal parameter list**. These formal parameters act as variable declarations for the function with the additional feature that they receive initial values from the arguments when the function is called; the first parameter gets the value of the first argument, the second parameter the value of second argument, and so on. The formal parameters in a function definition behave in the same manner as automatic variables, and their scope is limited to the function itself. The names in this list are the names used within the function body to access these values.

The body of the function is defined, as with `main()`, within brackets, { and } and consists of the variable declarations for the block followed by the executable statements to perform the subtask of the function. In our case, we declare variables `regular`, `overtime`, and `total` which are called **local variables** because their scope is local, i.e. limited to within the function. We then calculate regular pay, overtime pay and total pay as before, but we use the formal parameter names and the names of the local variables in our computations. Finally, since a function can

return only one value, we return only the value of total pay:

```
    return total;
```

The above `return` statement returns the *value* of the variable, `total`, to the calling function. In general, a `return` statement can be used to return the value of any expression. When the `return` statement is executed, the program control returns immediately to the calling function where the function call *evaluates* to the returned value.

When a function is first written, it is a good practice to include debug statements in the function definition showing the name of the function entered, the values of the parameters received, and the value returned by the function. When the program is run, these debug statements will produce a trace of all function calls and returns and as such are invaluable for debugging, particularly when a program uses many functions. We have included `printf()` statements for this purpose in the code for `calc_pay()` shown in the figure.

The above function, together with `main()` in the file pay5.c, forms a complete program which may be compiled and executed. A sample session shown below is similar to the one for `pay4.c`. The only change is that `calc_pay()` calculates and returns total pay, whereas in `pay4.c` total pay was calculated in `main()`.

```
    ***Pay Calculation***

    Type ID Number, zero to quit:  123
    Hours Worked:  20
    Hourly Rate:  7.5

    debug:entering calc_pay():  hours = 20.000000, rate = 7.500000
    debug:returning from calc_pay():  150.000000

    ID Number = 123
    Hours Worked = 20.000000, Rate of Pay = $ 7.50
    Pay = $ 150.00

    Type ID Number, zero to quit:  0
```

The debug printing clearly shows argument values at entry to `calc_pay()` and the returned value. If there are any bugs in a function, such debug printing helps detect and remove them.

3.2.1   Passing Data to and from Functions

As we can see from the above description, and also in Figure 3.5, information is passed to a function as arguments specified in the calling expression. This information is received by the function in the cells reserved for the formal parameters. In our case, the values of `hours_worked`
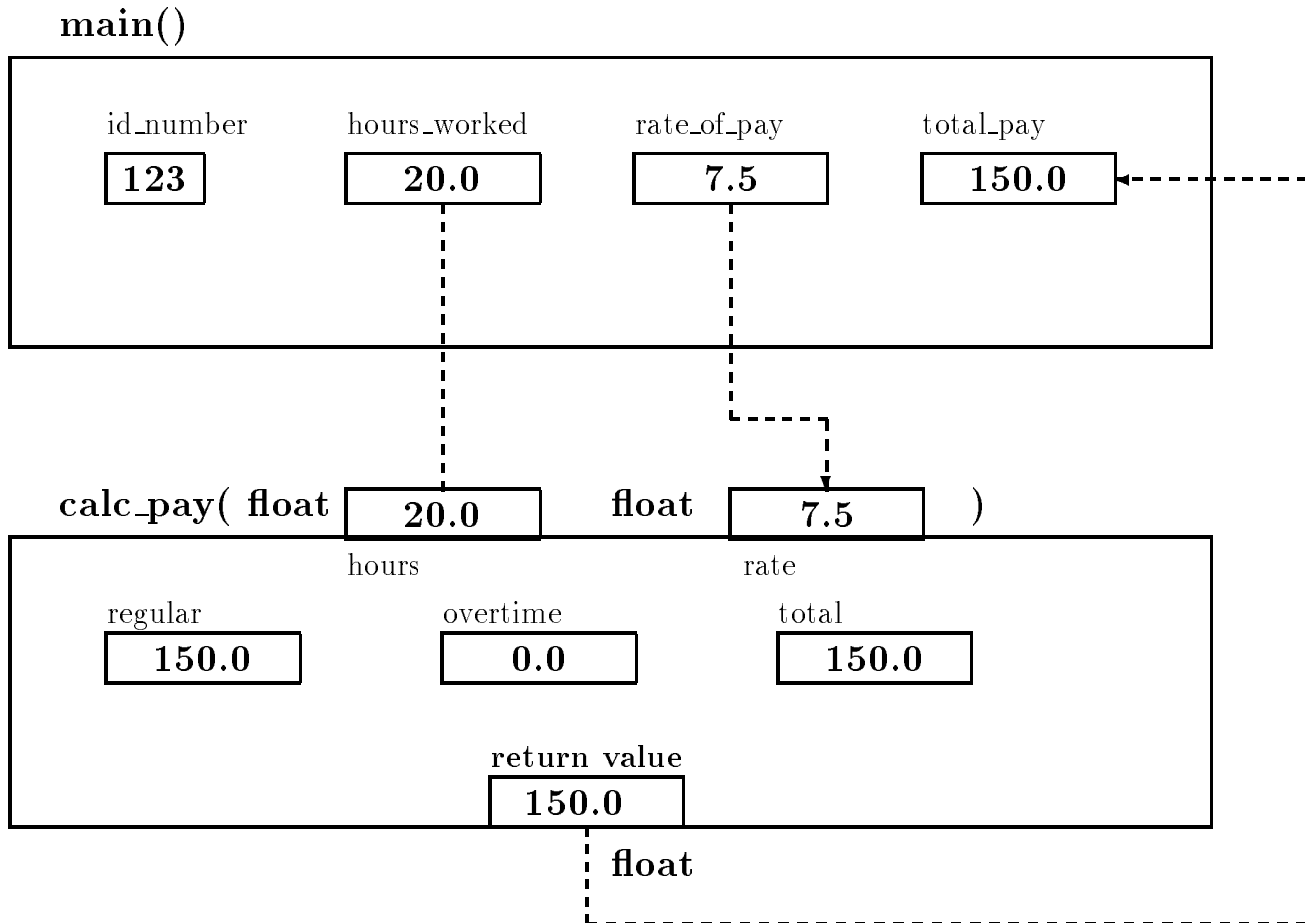
main()

Figure 3.5: Function Call Trace

and **rate_of_pay** (the arguments of the call) are copied to the cells called **hours** and **rate** within the function **calc_pay()**. Remember, these names are only known internally to the function. All that **main()** sees of the function is a *black box* as was shown in Figure 3.3.

The names of the formal parameters are arbitrary. For example, **calc_pay()** may be defined with any names for formal arguments:

```
float calc_pay(float x, float y)
{
    if (x > REG_LIMIT) ...
}
```

or,

```
float calc_pay(float hours_worked, float rate_of_pay)
{
```

```
        if (hours_worked > REG_LIMIT) ...
    }
```

As long as the function uses the formal parameters names internally for computations, the function definitions behave the same. In the last case, even though the formal parameters have the same names as variables defined in `main()`, they represent distinctly different variables, as shown in Figure 3.5. In summary, the scope of automatic variables defined in a block is local to that block, i.e. the objects can be directly accessed by name only within that block and in blocks nested within it.

As we stated earlier, the arguments in a function call can be any valid expressions. Only the values of the argument expressions are passed to the called function. For example, these are valid function calls:

```
    printf("Pay = %f\n", hours_worked * rate_of_pay);
    printf("Pay = %f\n", calc_pay(hours_worked, rate_of_pay));
    calc_pay(hours_worked, rate_of_pay * 1.10);
```

The argument in the first `printf()` call is a product expression. The result of evaluating that expression is passed to `printf()`. The second statement uses an argument that is itself a function call. The function call evaluates to a value which is then passed to `printf()`. The second argument in the last statement is an expression whose value is passed to `calc_pay()`.

Information is returned from a function using the `return` statement which can also return the value of any valid expression. The syntax of the `return` statement is:

```
    return <expression>;
```

For example, we could have combined the last two statements in the function definition of `calc_pay()`:

```
    return regular + overtime;
```

where `calc_pay()` would then return the value of the expression `regular + overtime`.

When writing functions, tools such as shown in Figure 3.5 can be very useful in tracing the behavior of the function. Another way to check a function for bugs is to manually trace its execution with representative values for the formal parameters. Figure 3.6 shows such a trace for `calc_pay()`. Note: the variables `hours` and `rate` (the formal parameters) receive values during the function calls. Other local variables get values as the function is executed.

In our payroll program, the overall logic can be made even more apparent if functions are used to get the input data and to print the results. The driver, i.e. `main()`, can then follow the overall logic and use function call statements to get the data, calculate the pay, and print the results. A function that prints data is simple to write. Writing a function that reads data is somewhat more involved. We will delay writing such functions until Chapter 6.

```
                                        hours  rate   regular overtime   total
float calc_pay(float hours, float rate)  20.0   7.5     ??       ??        ??
{    float regular, overtime, total;

printf("debug:entering calc_pay(): hours = %f, rate = %f\n",
            hours, rate);
    if (hours > REG_LIMIT) {
        regular = REG_LIMIT * rate;
        overtime = OT_FACTOR * rate *
                    (hours - REG_LIMIT);
    }
    else {
        regular = hours * rate;          20.0  7.5    150.0    ??        ??
        overtime = 0;                    20.0  7.5    150.0    0.0       ??
    }
    total = regular + overtime;          20.0  7.5    150.0    0.0      150.0
printf("debug:returning from calc_pay(): %f\n", total);
    return total;
}
```

Figure 3.6: Trace for `calc_pay()`

3.2.2 Call by Value and Local Variables

This section reviews and formalizes several features of variables that we have already encountered. We know that direct access of objects is performed by using variable names in expressions. The use of a variable on the left side of an assignment operator stores a new value in that object; the use of a variable anywhere else retrieves the value of the object. Objects defined in one function are not directly accessible to other functions. A calling function passes values of arguments to a called function. Only the values of these arguments, and NOT the arguments themselves, are available to the called function. The values of the arguments are stored in the parameters, and only the called function has access to these parameters. When called functions have access only to argument values, and not to arguments themselves, the function calls are termed **call by value**. In C, all function calls are call by value. It is impossible for a called function to have direct access to an object defined in the calling function. Let us examine the implications. Consider a program that uses a function to increment the value of an argument.

```
    /*   File: incr.c                          Program Trace
         Program demonstrates call by value.        x
    */
    #include <stdio.h>
    main()
    {    int x;                                      ??
         int incr(int n);
```

```
    printf("***Call by Value***\n");
    x = 7;                                              7
    printf("Original value of x is %d\n", x);           7
    printf("Value of incr(x) is %d\n", incr(x)); 7

    printf("The value of x is %d\n", x);                7
}



/* Function increments n */                             n
int incr(int n)                                         7
{
    n = n + 1;                                          8
    return n;                                           8
}
```

Compiling and executing this programs gives the following sample session:


```
  ***Call by Value***
  Original value of x is 7
  Value of incr(x) is 8
  The value of x is 7
```

The program trace shows that x in main() is assigned a value of 7 prior to a function call to incr() which increments its parameter to 8 and returns that value. After the function call, the value of x in main() is still 7, unchanged because only the *value* of x is passed to incr(). It was the cell, n, in incr() that was incremented as seen in Figure 3.7

We see that a called function cannot directly change the value of an object defined in the calling function. This is true even if the formal parameter in incr() were called x. Formal parameters represent new and distinct objects unrelated to any other objects defined elsewhere.

The variables declared at the beginning of a block (e.g. a function body) have all been of a storage class called **automatic**. This means that these variables are automatically created and destroyed each time the function is executed. When the execution of a function begins, the variables declared at the beginning of the function block as well as the formal parameters are created, i.e. memory cells for these variable names are allocated. When the execution of a function is completed (e.g. when a **return** statement is executed), the memory allocated for these variables is freed, i.e. these variables and their values no longer exist.

Automatic variables can be defined at the beginning of any block within the primary function block and exist only in the block in which they are defined. Memory for automatic variables declared in a block is allocated when the block is entered, and freed when the block is exited.
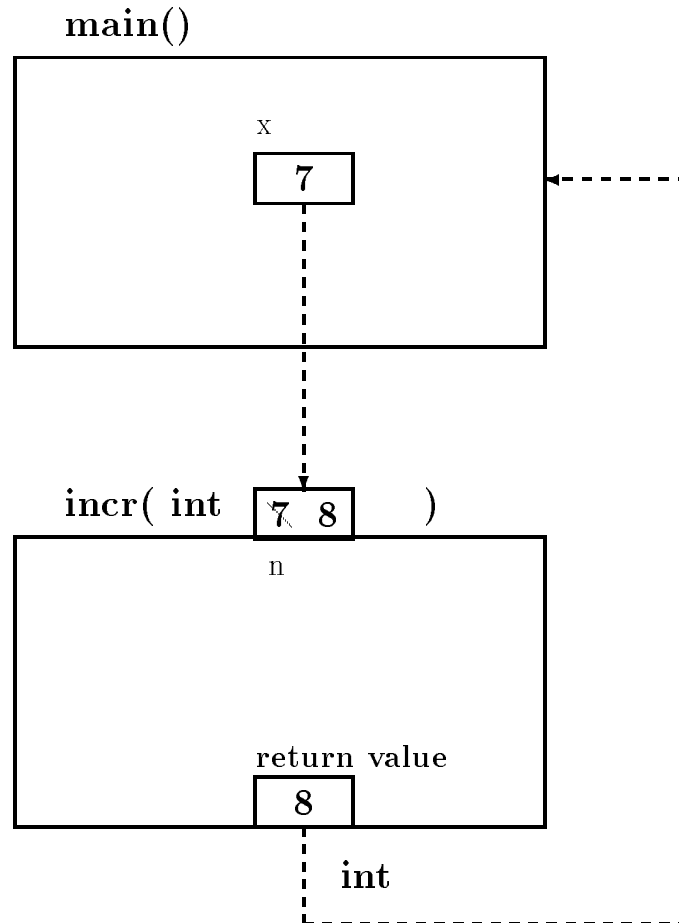
Figure 3.7: Call by value variable allocation

The **scope** of a variable is that part of the program where the variable is visible, i.e. where the variable can be accessed directly by name. The scope of automatic variables is local to the block in which they are defined as well as any blocks nested within it. Automatic variables are frequently referred to as **local variables**, since their scope is *local*.

A variable of automatic storage class can be explicitly defined in a declaration by preceding it with the keyword `auto`. Thus, the following declarations declare automatic variables:

```
auto int x, y;
auto float r;
```

If no storage class is specified in a declaration, automatic storage class is assumed by default. In all of our programs, so far, declarations have been for automatic variables by default. In general, most variables used in programs are automatic, and the default declaration without the keyword `auto` is a standard practice. Other storage classes will be discussed in Chapter 14. Until then, we will use only automatic variables.

As we stated before, a declaration only allocates a memory cell and associates the name with the cell; the value in that cell is, in general, unknown. However, it is possible to specify initial values of automatic variables in the declaration statements. Examples include:

```
int x = 5 * 2;
int y = isquare(2 * x);
float z = 2.8;
```

The first declaration initializes x to 10, and the second initializes y to the value returned by the function call isquare(2 * x). If the function isquare() returns the square of its argument, then y in this case, is initialized to 400, i.e. the square of 2 * x. Finally, the last declaration initializes the variable z to the value 2.8.

The syntax for a declaration statement with initialization is:

<type_specifier><var_name> [=<init_expr>] [,   <var_name> [=<init_expr>]...];

The declaration allocates memory for each <var_name> of a type indicated by <type_specifier>, and initializes the variable to the value of the initializer expression, <init-expr>. The initializer expression can be any C expression including function calls.

Consider the following example in which automatic variables are declared in nested blocks:

```
/*    File: auto.c
      Program shows declarations of automatic variables in nested
      blocks. Scope of automatic variables is the block in which they
      are defined.
*/
main()
{                                  /* outer block */
    auto int x = 10, z = 15;       /* x and z are allocated and initialized */

    printf("***Automatic Variables and Scope***\n\n");
    {                              /* inner block */
        int x = 20, y = 30;        /* new variables x and y are allocated */
                                   /* only the new x can be accessed */
        printf("In the inner block: \n");
        printf("x = %d, y = %d, z = %d\n",
                x, y, z);          /* new x and y, and z are printed */
    }                              /* new x and y are freed */
    printf("In the outer block:\n");
    printf("x = %d, z = %d\n", x, z); /* only the old x can be */
                                      /* accessed in the outer block.*/
    /* printf("y = %d\n", y);          error: y is not visible here. */
}
```
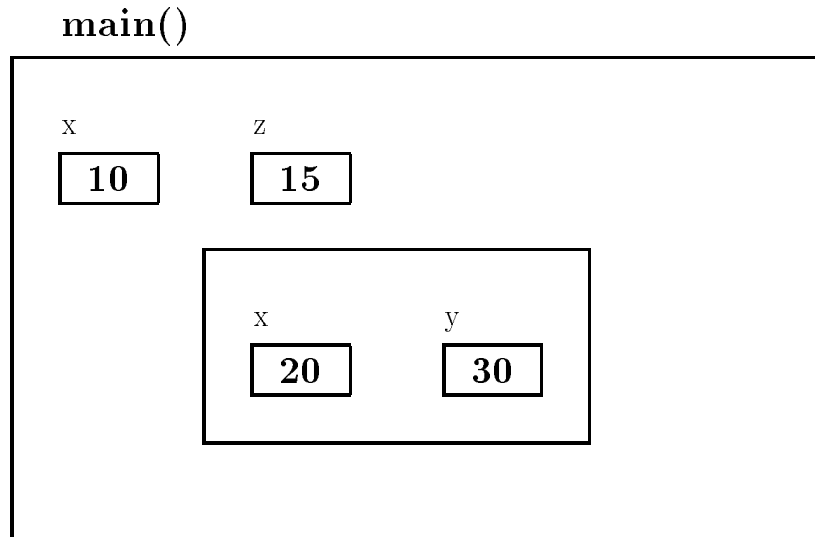
main()

Figure 3.8: Local Variables in Blocks

The program contains an outer block, which is the function body for `main()`, and an inner block. The scope rules say that an inner block can access variables declared within it plus any variables declared in an enclosing block. However, if the same variable name is used in an inner and an outer block, the local variable in the inner block is accessed. The outer block cannot access variables defined in an inner block.

In the example, variables `x` and `z` are declared in the outer block and assigned values. The outer block can access only these variables. Variables `x` and `y` are declared in the inner block and assigned values. The inner block can access the variables `z`, `y`, and that `x` which is defined in the inner block. As shown in a comment, if the outer block tried to access `y`, a compile time error would occur. This behavior can be seen in Figure 3.8. The allocation of storage is shown when the program is executing within the inner block as can be seen by the nested box containing `x` and `y`. When this block is completed, the inner box, and all variables inside, is freed. A sample output of the program shows the results:

```
***Automatic Variables and Scope***

In the inner block:
x = 20, y = 30, z = 15
In the outer block:
x = 10, z = 15
```

It is also possible to qualify an automatic variable as a constant using the keyword `const`. A `const` qualifier allows initialization of a variable but the variable may not be otherwise changed within the program. Here is an example:

```
const int x = 100;
```

In the above case, x is initialized to 100 and qualified as a constant. Its value may not be changed elsewhere in the program, e.g. in an assignment statement. Constant qualifiers are used to ensure that certain variable values are not altered by oversight.

Let us consider a somewhat more meaningful example that declares a variable in an inner block. The task is to swap values of two objects, x and y. We need a temporary variable to save one of the values; otherwise, assigning the value of y to x would overwrite the original value of x. We can declare the temporary value in an inner block.

```
/*   File: swap.c
     This program swaps values of two objects. It defines and uses a
     temporary variable in an inner block.
*/
#include <stdio.h>
main()
{    int x = 10, y = 20;

     printf("***Swap Values***\n\n");
     printf("Original values: x = %d, y = %d\n", x, y);
     {    int temp;

          temp = x;
          x = y;
          y = temp;
     }
     printf("Swapped Values: x = %d, y = %d\n", x, y);
}
```

Here is the output of the program:

```
***Swap Values***

Original values:  x = 10, y = 20
Swapped Values:  x = 20, y = 10
```

Defining variables in blocks other than a primary function block is not recommended unless there are good reasons for it. In the above example, a temporary variable is declared closest to its use and has no logical role in the rest of the program. When a function uses many variables, declaring variables closest to their use may make it easier to understand the program behavior. For the most part, we will declare all variables at the beginning of primary function blocks.

The formal parameters of a function are also variables that are automatically allocated during a function call, and into which the argument values are passed. Their values, just like those of any other variables, may be changed in the function. The scope of the formal parameters is the body of the function, i.e. the scope is local to the function body.

3.3 Coding Programs for Readability

In the previous sections we have seen how to organize programs modularly, beginning with the algorithm, and carrying that organization into the code using functions. This is a form of *information hiding*, i.e. the details of performing a particular operation are hidden from the more abstract steps of the algorithm. Here we are hiding *ideas* or *abstractions* at the algorithm level. Another form of information hiding at the source code level is described in this section; namely hiding the details of the *syntax* of the language in order to make the source code more readable.

3.3.1 The C Preprocessor

We have already seen that in order for a program to be run, it must be compiled, i.e. translated from the C language to the machine language of the computer being used. This compilation process takes place in several steps; the source code is read from the file, checked for proper syntax, and analyized for the meaning of the statements in the code. The proper machine language steps to perform these statements can then be generated (and optimized) and then linked with other functions to produce the executable file. At the beginning of this entire process, standard C compilers provide an additional step called the **preprocessor**. The source code is read from the file and given to the preprocessor where it is translated into a modified source code file which is then given to the compiler proper for translation to machine language. The transformations performed by the preprocessor are directed by lines in the original source file called **compiler directives**. All such lines begin with the # character as the first non-white space character on the line and are of one of three types of directives: macro definitions, file inclusion, and conditional compilation. Each of these are discussed in the following sections.

3.3.2 Macros

In Chapter 2 we introduced the `define` compiler directive which defines symbolic names for strings of characters. Such a string of characters can be arbitrary, for example a sequence of characters representing a numeric constant. These names can then be used anywhere in the program instead of the string itself. The C preprocessor replaces these symbolic names with the specified strings prior to compiling the program. We have seen examples where using names for arbitrary strings makes it easy to change all occurrences of these names by merely changing the definitions. It also makes for easier reading and debugging of programs by allowing the programmer to use a name which has some meaning rather than some "magic number".

The definition is called a **macro** and the preprocessor performs a **macro expansion** when it substitutes the string for the name. A macro definition takes the form:

```
#define <symbol_name> <substitution_string>
```

The macro names follow the same rules as identifiers, however, a common convention observed

by most C programmers is to name macros in all upper case to distinguish them from program variables. No quotation marks are used to delimit the string, nor is the directive terminated by a semi-colon. Instead, the string extends to the end of the line (an escape character, \, can be used to continue the string on the next line). For example, the following are macro definitions:

```
#define     PI          3.14159
#define     SIZE        1000
#define     RSQUARED    radius * radius
#define     AREA        PI * RSQUARED
#define     LONG        This is a very long macro \
definition we continued to the next line
```

When directives such as these appear in the source file, then the macros are said to have been defined. We have defined macros for the symbols PI, SIZE, RSQUARED, AREA and LONG. With the above definitions, the defined names may be used anywhere in program statements. The preprocesser generates the expanded source code by string replacement, for example:

```
        Original code              Expanded code after preprocessing


    circum = 2 * PI * radius;      circum = 2 * 3.14159 * radius;
    y = x + SIZE;                  y = x + 1000;
    printf("SIZE = ",SIZE);        printf("SIZE = ",1000);
    AREA;                          3.14159 * radius * radius;
```

As can be seen, the preprocessor replaces the macro name with the specified replacement string in the entire source file following the definition. The substitution is not made if a macro name, occurs in double quotes as in the format string in the printf() statement shown above.

The scope of the macro definition is the entire source file following the definition line. The definitions may be removed at any point in the program by a directive #undef, for example:

```
    #undef SIZE
```

The above directive makes the preprocessor "forget" the previous definition for SIZE. If desired, a new definition may be specified for SIZE at this point. It is a common practice to put macro definitions at the top of the source file, unless the old definitions are removed at some point in the source file and new definitions are specified:

```
    #define SIZE 40            /* SIZE is define to be the string 40 */
    ...
    #undef SIZE                /* SIZE is undefine */
    #define SIZE 100           /* SIZE is defined to be 100 */
```

Identical definitions for identifiers may appear in a file without causing any problems; however, two different definitions for an identifier represent an error.

```
#define SIZE 40
#define SIZE 40      /* OK */
#define SIZE 100     /* ERROR */
```

The only way to make a new definition for an identifier is to first undefine it, i.e. remove its first definition.

Macros with Arguments

Macro definitions may also have formal parameters which are replaced by the actual arguments given in the macro call. This is similar to parameters in function calls; however, macro arguments are treated as *strings of characters* and are substituted for parameters by the preprocessor; no evaluation takes place. Consider the example:

```
#define READ_FLT(fvar)   scanf("%f", &fvar)
```

The macro encapsulates the expression for reading a `float` number, i.e. a macro call is replaced by a string that represents a correct `scanf()` function call to read a `float` number into an object passed to the macro. The actual argument in a macro call replaces `fvar` in the replacement string. In other words, every time the macro is called, the expanded code is substituted literally except that `fvar` in the definition is replaced by the argument given in the actual call. Here are some examples of macro calls with parameters together with the expanded code:

```
        macro call              Expanded Code


    READ_FLT(x);            scanf("%f", &x);
    READ_FLT(rate);         scanf("%f", &rate);
```

Macro calls in these cases expand to C statements. Such calls are said to expand to `in-line code`, because the resulting code represents statements in the source code. These types of macro calls can be used in place of function calls, for example, instead of writing a function to square a number, we can define a macro:

```
#define SQ(x)        (x * x)
```

We can use such a macro in any expression, e.g.,

```
y = SQ(radius);
printf("Square of %d is %d\n", radius, SQ(radius));
```

However, remember, macro calls are *substitutions*, and macro parameters are neither evaluated nor checked for data type consistency. Therefore, proper placement of parentheses is important in macro definitions. For example, consider the following macro call and expanded code:

```
SQ(x+y)
```

expanded becomes

```
(x + y * x + y)
```

The expanded code is not the square of `(x + y)`, as we would expect. By precedence rules, it is a sum of three terms, `x`, `y * x`, and `y`. A proper definition of a macro for square should be:

```
#define SQ(x)  ((x) * (x))
```

With this definition,

```
SQ(x+y)
```

will expand correctly to

```
((x + y) * (x + y))
```

Here is a simple example program:

```
/* File: macro.c */
#define READ_FLT(fvar)   scanf("%f", &fvar)
#define PI     3.14159
#define SQ(x)  ((x) * (x))

main()
{
    float radius;

    printf("Type Radius: ");
    READ_FLT(radius);
    printf("Area of a circle with radius %6.2f is %6.2f\n",
            radius, PI * SQ(radius));
}
```

The output of a sample run is:

```
Type Radius:   10
Area of a circle with radius 10.00 is 314.15
```

Why use macros with arguments when functions will serve the same purpose? The advantage is practical, NOT logical. When a function is called, there is a certain amount of run time overhead, i.e. extra time needed during execution. The overhead comes from passing arguments, transferring control, returning a value, and returning control. If a function is called just a few times, the overhead is negligible. However, if a function is used numerous times, e.g. in a loop executed many times, then the overhead can become significant.

A macro on the other hand has no run time overhead. It is expanded at compile time into in-line code which has no overhead at run time. If execution time for a program is a problem because of a frequently used routine, then writing a macro for that routine makes good sense, as long as the operation can be simply expressed as a macro.

An Example Program

Let us look at another example program to make use of these new facilities.

Task

Read a set of high temperature readings for some number of days and to count the number of nice days, bad days, and the average temperature for the period. Nice days are those days whose temperature falls within some "comfort zone".

The high level algorithm for this task is straight forward;

```
prompt the user and read first temperature
while there are more days to read
        process one day's temperature
        accumulate total temperature
        read the next temperature
print results
```

With this algorithm, we next consider what information we will be working with in this program. We read daily temperatures, so we will need a variable for that, and variables to count the number of nice and bad days. Since we compute the average temperature, we accumulate the total of all the daily temperatures, so we need a variable for that. Next we consider how we will implement the algorithm using functions to hide details. For example, the step to print results, printing the number of nice and bad days as well as computing and printing the average temperature can be done in a function, `print_results()`, which is given the number of nice days, bad days, and the cumulative total of temperatures. The step of processing one day's temperature is another candidate; however, this step involves updating our counts of nice and bad days. Since, as we

have seen, functions cannot access variables local to main, we refine our algorithm to fill in some
of the details of this step:

```
prompt the user and read first temperature
while there are more days to read
        if it's a nice day, count a nice day
        otherwise count a bad day
        accumulate total temperature
        read the next temperature
print results
```

We can use a function to test if a day is nice, thus hiding the details of this operation. We are
now ready to write the code for main() as shown in Figure 3.9. It should be noted we have made
an additional design decision here; we use a zero value for the temperature read in as the loop
termination. Also not that we have provided prototype statements for our functions, nice_day()
and print_results(). This is sufficient information about these functions when considering the
logic of main(). (We have specified the return value of print_results as type int, but the function
has no real meaningful return value).

We next turn out attention to the function, nice_day(). This function is given the temperature
and should return True if this qualifies as a nice day, and False otherwise. The task specified that
the temperature of a nice day is to fall within some "comfort zone", i.e. not too cold and not too
hot. We can write the algorithm for this function from this information:

```
if temperature is too cold, return False
if temperature is too hot, also return False
otherwise, this is a nice day, return true
```

We choose to implement the too cold and too hot tests using macro:

```
#define    TOO_COLD      80
#define    TOO_HOT       90

#define    HOT_DAY(t)    ((t) > TOO_HOT)
#define    COLD_DAY(t)   ((t) < TOO_COLD)
```

Coding of the function is straight forward. Similarly, for print_results(), the algorithm is:

```
print number of nice days and bad days
if there are any days counted
        compute the average temperature
        print the average temperature
```

```
/*   File: niceday.c
     Programmer: Programmer Name
     Date: Current Date
     This program counts the number of nice days in a set of high
     temperature data.
*/

int nice_day(int temp);
int print_results(int nice, int bad, int temp_sum);

main()
{ /*  declarations  */
   int temperature,          /*  daily temperature  */
       total = 0,            /*  cumulative total   */
       num_nice_days = 0,
       num_bad_days  = 0;

   /*  print title and prompt  */
   printf("***Count Nice Days***\n\n");
   printf("Type daily high temperature readings (0 to quit): ");

   /*  read the first temperature  */
   scanf("%d", &temperature);
   while (temperature != 0)  {

           /*   process one temperature  */
           if ( nice_day(temperature))
               num_nice_days = num_nice_days + 1;
           else
               num_bad_days = num_bad_days + 1;
           /*  accumulate  total of temperatures  */
           total = total + temperature;

           /*   read next temperature  */
           scanf("%d", &temperature);
   }

   print_results(num_nice_days, num_bad_days, total);

}
```

Figure 3.9: Driver for niceday.c

```
/*    File: niceday.c (continued) */

#define    TRUE              1
#define    FALSE             0

#define    TOO_COLD     80
#define    TOO_HOT      90

#define    HOT_DAY(t)     ((t) > TOO_HOT)
#define    COLD_DAY(t)    ((t) < TOO_COLD)

#define    ANY_DAYS(n,b) (((n) + (b)) > 0)

/*  Function to test for a nice day given the temperature        */
int nice_day(int temp)
{
   if( COLD_DAY(temp))  return FALSE;

   if( HOT_DAY(temp))   return FALSE;

   return TRUE;

}

/*  Function to print results given number of nice and bad days  */
/*     and total of temperatures                                 */
int print_results( int nice_days, int bad_days, int total)
{
   float average_temp;

   printf("There were %d nice days and %d bad days\n",
                        nice_days, bad_days);

   if ( ANY_DAYS( nice_days, bad_days))  {
      average_temp = (float) total / (float) (nice_days + bad_days);
      printf("The average temperature for %d days was %f\n",
                        nice_days +  bad_days, average_temp);
   }
}
```

Figure 3.10: Functions for niceday.c

The resulting code for these functions is shown in Figure 3.10

Compiling and executing this program with some sample data produces the following sample session:

```
***Count Nice Days***

Type daily high temperature readings (0 to quit):  83
85
88
92
94
86
82
80
79
0
There were 6 nice days and 3 bad days
The average temperature for 9 days was 85.444443
```

3.3.3   Including Header Files

The second feature provided by the preprocessor allows us to break our source files into smaller pieces to be reassembled at compile time. Using functions to hide details of algorithms and macros to hide the syntax and "magic numbers" to make our programs more readable often results in many function prototype statements and macro definitions at the beginning of source code files. These may also be hidden in separate files, and included in the source file by the preprocessor. The files containing this information to be included are called **include files** or **header files**, and by convention, are named with a .h extension on the file name. Header files are also often used to provide common macro definitions and prototype statements that may be useful in may programs (or as we shall see later, in many files making up a single program). An example of the later case are the standard library functions provided in C; the prototype statements for these functions should be available to any program which chooses to use the functions. In many of our programs so far, we have used the library functions `printf()` and `scanf()`. Where are the prototypes for these? As well as providing the code for library functions, all standard C implementations provide a set of .h files with this information. The file `stdio.h` contains the prototypes and macros needed to use the I/O library. (We have not needed this file before because the compiler will make assumptions about functions if prototypes are not provided. Sometimes these assumptions are "safe", but often they are not. It is a good idea, from now on, to include `stdio.h` in any program using the I/O library).

The statements and directives in an include file are inserted in a source file when the preprocessor encounters an `#include` directive in the original source file. To include `stdio.h` the directive is:

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

The angle brackets, < and >, surrounding the filename indicate to the preprocessor that the file, `stdio.h`, is to be found in "the usual place" where standard header files are kept on the system (this is system dependent), and its contents placed in the source code in place of the `#include` directive. Any other directives within the included file (such as `#define` or other `#include` directives) are also processed at this time.

Besides the standard header files, as a programmer you can create and include your own header files for your programs. For example, in our `niceday.c` program, we defined macros for `TRUE` and `FALSE`. These macros are very common in many programs, so it would be convenient if we could enter those definitions in a single header file and simply include that header file in any program the uses those macros. This header file might be called `tfdef.h` and contain:

```
/*   File: tfdef.h
     Programmer: Programmer Name
     This file contains the definitions of TRUE and FALSE
*/



#define   TRUE            1
#define   FALSE           0
```

To include these definitions in a `.c` source file, use the directive:

```
    #include "tfdef.h"
```

Notice in this instance that the file name is surrounded by double quote, `"`, characters rather than the angle brackets used before. This syntax tells the preprocessor that the header file is to be found in the same directory as the `.c` source file currently being processed.

Again, in our nice day program, all of the other macro definitions and prototypes relating just to this program may also be placed in a header file, say `niceday.h`:

```
/*   File: niceday.h
     Programmer: Programmer Name
     This file contains the definitions of macros and prototypes
     for functions used by the niceday program.
*/



#define   TOO_COLD        80
#define   TOO_HOT         90
```

```
#define   HOT_DAY(t)    ((t) > TOO_HOT)
#define   COLD_DAY(t)   ((t) < TOO_COLD)

#define   ANY_DAYS(n,b) (((n) + (b)) > 0)

int nice_day(int temp);
int print_results(int nice, int bad, int temp_sum);
```

and replaced in `niceday.c` with:

```
    #include "niceday.h"
```

Thus, the beginning of `niceday.c` has been reduced to:

```
/*  File: niceday.c
    Programmer: Programmer Name
    Date: Current Date
    This program counts the number of nice days in a set of high
    temperature data.
*/

#include <stdio.h>
#include "tfdef.h"
#include "niceday.h"

main()
{  ...
```

Notice we include `stdio.h` at the head of the source file. Its contents are available for use by the entire source file. We also declare the function prototypes for `nice_day()` and `print_results()` in the file `niceday.h` outside `main()`. A declaration outside a function is called an external declaration. The scope of an external declaration is the entire file from the point of the declaration; i.e. all code that follows the external declaration can use the declared item. Since `stdio.h` is included outside `main()`, the declarations for `scanf()` and `printf()` are also external. External declaration of functions is convenient since it avoids repeated declarations of the same function. On the other hand, external declarations of variables leads to poorly structured programs and destroys modularity of functions. External declarations of variables is strongly discouraged.

In summary, the syntax of the `#include` directive is:

```
    #include <filename>
    #include "filename"
```

with the semantics that the contents of the file, `filename`, is to be inserted in the source file in place of the `#include` directive. (Note: here the angle brackets are part of the syntax of the directive). Other directives in the included file are also processed. In the first form of the directive, the header file is searched for in the "usual place" for system header files, and in the second case, it is to be found in the current directory. The advantages of using the `#include` directive are twofold:

1. Information such as macro definitions and prototype statements that are useful in multiple program files need only be entered in a single place and then included where needed. This also facilitates changes; the change need be made only in a single place.

2. Details of macro definitions and prototypes are hidden from the view of the reader, thus alleviating clutter and information overload and allowing a reader of the program to concentrate on the logic of the code itself.

3.3.4   Conditional Compilation

The third useful facility provided by the preprocessor is **conditional compilation**; i.e. the selection of lines of source code to be compiled and those to be ignored. While conditional compilation can be used for many purposes, we will illustrate its use with debug statements. In our previous programming examples, we have discussed the usefulness of `printf()` statements inserted in the code for the purpose of displaying debug information during program testing. Once the program is debugged and accepted as "working", it is desirable to remove these debug statements to use the program. Of course, if later an undetected bug appears during program use, we would like to put some or all debug statements back in the code to pinpoint and fix the bug. One approach to this is to simply "comment out" the debug statements; i.e. surround them with comment markers, so that if they are needed again, they can be "uncommented". This is a vast improvement over removing them and later having to type them back. However, this approach does require going through the entire source file(s) to find all of the debug statements and comment or uncomment them.

The C preprocessor provides a better alternative, namely conditional compilation. Lines of source code that may be sometimes desired in the program and other times not, are surrounded by `#ifdef`,`#endif` directive pairs as follows:

```
#ifdef DEBUG
printf("debug:x = %d, y = %f\n", x, y);
     ...
#endif
```

The `#ifdef` directive specifies that if `DEBUG` exists as a defined macro, i.e. is defined by means of a `#define` directive, then the statements between the `#ifdef` directive and the `#endif` directive are retained in the source file passed to the compiler. If `DEBUG` does not exist as a macro, then these statements are not passed on to the compiler.

Thus to "turn on" debugging statements, we simply include a definition:

```
#define DEBUG   1
```

in the source file; and to "turn off" debug we remove (or comment) the definition. In fact, the replacement string of the macro, `DEBUG` is not important; all that matters is the fact that its definition exists. Therefore,

```
#define DEBUG
```

is a sufficient definition for conditional compilation purposes. During the debug phase, we define `DEBUG` at the head of a source file, and compile the program. All statements appearing anywhere between `#ifdef` and matching `#endif` directives will be compiled as part of the program. When the program has been debugged, we take out the `DEBUG` definition, and recompile the program. The program will be compiled excluding the debug statements. The advantage is that debug statements do not have to be physically tracked down and removed. Also, if a program needs modification, the debug statements are in place and can simply be reactivated.

In general, conditional compilation directives begin with an if-part and end with an endif-part. Optionally, an else-part or an elseif-part may be present before the endif-part. The keywords for the different parts are:

> *if-part*: if, ifdef, ifndef
> *else-part*: else
> *elseif-part*: elif
> *endif-part*: endif

The syntax is:

```
# <if-part>
        <statements>
[ # <elseif-part>
        <statements> ]
[ #<else-part>
        <statements> ]
#<endif-part>
```

If the if-part is True, then all the statements until the next <else-part>, <elseif-part> or <endif-part> are compiled; otherwise, if the <else-part> is present, the statements between the <else-part> and the <endif-part> are compiled.

We have already discussed the keyword ifdef. The keyword ifndef means "if not defined". If the identifier following it is NOT defined, then the statements until the next <else-part>, <elseif-part> or <endif-part> are compiled.

The keyword if must be followed by a constant expression, i.e. an expression made up of constants and operators. If the constant expression is True, the statements until the next else-part, elseif-part or endif-part are compiled. In fact, the keyword ifdef is just a special case of the if form. The directive:

```
#ifdef ident
```

is equivalent to:

```
#if defined ident
```

We can also use #if to test for the presence of a device, for example, so that if it is present, we can include an appropriate header file.

```
#if DEVICE == MOUSE
    #include mouse.h
#endif
```

Here, both DEVICE and MOUSE are assumed to be constant identifiers.

The #elif provides a multiway branching in conditional compilation analogous to else ...if in C. Suppose, we wish to write a program that must work with any one of a variety of printers. We need to include in the program a header file to support the use of a specific printer. Let us assume that the specific printer used in an installation is defined by a macro DEVICE. We can then write conditional compilation directives to include the appropriate header file.

```
#if DEVICE == IBM
    #include ibmdrv.h
#elif DEVICE == HP
    #include hpdrv.h
#else
    #include gendrv.h
#endif
```

Only constant expressions are allowed in conditional compilation directives. Therefore, in the above code, DEVICE, IBM, and HP must be be defined constants.

The niceday Example Again

Using compiler directives is a convenience for the programmer and makes program source files easier to understand. One goal in understandable files is to make them small, the less a reader has to look at in trying to understand a program, the better. Good programming style includes

the hiding of details at the algorithm level with functions, at the source code level using macros, and at the source file level using header files and conditional compilation. One comment should be made about header files. The information stored in header files is meant to be directives and prototype statements, NOT code statements or function definitions. Also **DO NOT**:

```
#include "somefile.c"
```

The syntax of the `#include` directive allows these, but it is considered bad style. A final version of our file `niceday.c` using these compiler directives is shown in Figure 3.11.

3.4   Interacting with the Operating System

In the programs we have developed so far, we have used C library functions `scanf()` and `printf()` to perform the input and output for our programs. These library routines are simply functions that call on the facilities of the operating system to cause data to be the read from the keyboard and written to the screen. In this section we look in more detail at these features of the operating system.

3.4.1   Standard Files and EOF

In our payroll programs, we used a sentinel value of id number, namely 0, to indicate the end of input data. There are many instances when it is not possible to use a special sentinel value of input data to terminate the input. For example, suppose we wish to read a sequence of numbers and determine the largest of them. It is impossible to select any one number as a signal to terminate input since any selected number may be one of the valid numbers in our sequence and may appear before the entire sequence of numbers is exhausted. We need a way to indicate that the end of input is reached without entering any special value of input which may also be valid data.

C provides such mechanism to indicate the end of data input through the way it handles all input and output. All data read by a C program or written from a program can be considered to be simply a **stream** or sequence of characters, i.e. symbols we use to type or print information: alphabetic letters, digits, punctuations, etc. This stream of characters is called a **file** and is organized like any other file in the system. Three files, called **standard input**, **standard output**, and **standard error**, are predefined files available to all programs. By default, **standard input** is the keyboard, and **standard ouput** is the screen. The function `scanf()` reads data from the standard input file, and `printf()` writes data to the standard output file. Run time error messages are written to **standard error**, which is also the screen, by default.

The end of a file is indicated by a special marker which is an unusual character not commonly used for any other purpose. When input is typed at the keyboard, an end of file mark is indicated by what is called a control character. A control character is typed by pressing the *control key*, (CTRL), and pressing another key while keeping CTRL key pressed. For example, control-A is

```c
/*    File: niceday.c
      Programmer: Programmer Name
      Date: Current Date
      This program counts the number of nice days in a set of high
      temperature data.
*/

#include <stdio.h>
#include "tfdef.h"
#include "niceday.h"

main()
{  /*  declarations  */
   int temperature,          /*  daily temperature  */
       total = 0,            /*  cumulative total   */
       num_nice_days = 0,
       num_bad_days  = 0;

   /*  print title and prompt  */
   printf("***Count Nice Days***\n\n");
   printf("Type daily high temperature readings (0 to quit): ");

   /*  read the first temperature  */
   scanf("%d", &temperature);
   while (temperature != 0)  {

        /*   process one temperature  */
        if ( nice_day(temperature))
             num_nice_days = num_nice_days + 1;
        else
             num_bad_days = num_bad_days + 1;
        /*  accumulate  total of temperatures  */
        total = total + temperature;
#ifdef DEBUG
printf("debug: %d temps read, total = %d\n",
                  num_nice_days + num_bad_days, total);
#endif

        /*   read next temperature  */
        scanf("%d", &temperature);
   }

   print_results(num_nice_days, num_bad_days, total);

}
```

```
/*  Function to test for a nice day given the temperature     */
int nice_day(int temp)
{
   if( COLD_DAY(temp))  return FALSE;

   if( HOT_DAY(temp))   return FALSE;

   return TRUE;

}


/*  Function to print results given number of nice and bad days  */
/*    and total of temperatures                                  */
int print_results( int nice_days, int bad_days, int total)
{
   float average_temp;

   printf("There were %d nice days and %d bad days\n",
                       nice_days, bad_days);

   if ( ANY_DAYS(nice_days,bad_days))  {
      average_temp = (float) total / (float) (nice_days + bad_days);
      printf("The average temperature for %d days was %f\n",
                       nice_days +  bad_days, average_temp);
   }
}
```

Figure 3.11: Using Directives in `niceday.c`

entered by pressing CTRL and pressing A while keeping CTRL pressed. Control characters are displayed on screen or paper by a caret followed by a letter. For example, control-A is written as ˆA. The Control character entered on a keyboard to indicate an end of file is ˆD on most Unix machines and ˆZ on DOS machines. A keyboard file (stream) with an end of file keystroke is shown in Figure 3.12. Here, three lines of input are represented, followed by the end of file marker as if the user had typed:

> *89*
> *78*
> *0*
> *ˆD*

How does `scanf()` inform the calling function that an end of file has been reached? It does so by returning a special value to indicate an end of file. The function `scanf()` is just like any

8	9	\n	7	8	\n	0	\n	**EOF**	\n		
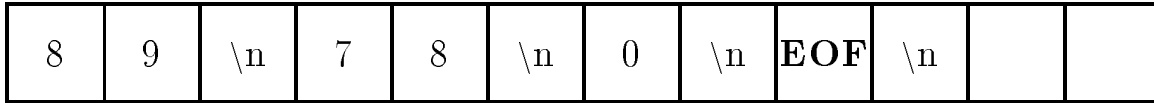
Figure 3.12: End of File Marker

other function in C; it has arguments passed to it and it returns a value. So far, we have simply ignored whatever value has returned. Normally, when `scanf()` reads data, it returns a value to indicate the number of data items read successfully. We can save this value returned by `scanf()` and examine whether all data items have been read. For example, consider:

```
flag = scanf("%d", &n);
flag = scanf("%d %f %d", &n, &y, &id);
```

Assuming that both the above statements read data items successfully, then the first `scanf()` will return 1 since it reads one decimal integer, and the second will return 3 since it reads three data items, two `int`'s and a `float`. We have not used this value so far, but we can use it to check if a correct number of items are read.

When `scanf()` detects the special end of file marker, it returns a value of either 0 or -1 (depending on implementation). The actual value returned is defined as a macro called `EOF` in the file `stdio.h`.

We can now write a loop that terminates when the end of standard input file is reached.

```
#include <stdio.h>
...
flag = scanf();
while (flag != EOF) {
    ...
    flag = scanf();
}
```

The value returned by `scanf()` is saved in the variable `flag`. The loop repeats until `flag` receives the value `EOF`. The above code is portable to any implementation since the correct value of `EOF` is defined in `stdio.h` in every implementation. We can now write a program that uses end of file to terminate reading of data.

Task

BIG: Find the largest absolute value in a sequence of integers typed in by the user. An end of file keystroke terminates the input.

The algorithm maintains the current largest absolute value. Each time a new number is read, the absolute value of the item read is compared with the largest value, and if necessary the largest value is updated. The algorithm uses a loop that is terminated when an end of file keystroke is typed. Here is the algorithm.

```
initialize largest to 0
read first integer, n
while there is still data
        compare absolute value of n and largest, update largest
        read next integer
print largest
```

We will need a function `absolute()` which takes an integer argument `n`, and returns its absolute integer value. Notice we initialize our largest absolute value to 0, since that is the smallest absolute value we can ever encounter. The entire program is shown in Figure 3.13 and a sample session is:

```
***Largest Absolute Integer***

Type integers, EOF to quit:  ^Z for DOS, ^D for Unix
-20
0
30
-60
^D
Largest absolute value = 60
```

In our program, `main()` first prompts the user to type integers, and it also tells the user how to terminate the input. It is best to assume that the user does not know how to press a keystroke for `EOF` (however, in the future we will omit this reminder and assume the user knows the correct `EOF` character). The prompt is written by:

```
printf("Type integers, EOF to quit: "
        "^Z for DOS, ^D for Unix\n");
```

Observe that the argument of `printf()` consists of two adjoining strings of characters, each in double quotes. When the compiler encounters two adjoining strings, it replaces them by a concatenated string, i.e. it joins them together into a single string:

```
"Type integers, EOF to quit: ^Z for DOS, ^D for Unix\n"
```

When a string gets too large, it is best to split it into two adjoining strings, since strings cannot be broken across lines.

```
/*   File: maxabs.c
     Programmer: Programmer Name
     Date: Current Date
     This program reads in a sequence of integers until an end of file.
     Among the numbers read, the program determines the largest absolute value.
*/
#include <stdio.h>
int absolute(int n);

main()
{    int largest = 0,
          n, flag;

     printf("***Largest Absolute Integer***\n\n");
     printf("Type integers, EOF to quit: ");
               "^Z for DOS, ^D for Unix\n");
     flag = scanf("%d", &n);

     while (flag != EOF) {
          if (absolute(n) > largest)
               largest = absolute(n);
          flag = scanf("%d", &n);
     }
     printf("Largest absolute value = %d\n", largest);
}

/* Function returns the absolute value of n */
int absolute(int n)
{
     if (n < 0)
          return -n;
     else
          return n;
}
```

Figure 3.13: Code for maxabs.c

After the prompt, `main()` reads the first integer. The while loop tests for the end of the input and compares the value of largest and the absolute value of the last number read, `n`. If necessary largest is updated, a new number is read, and so forth. The loop is terminated when an end of file character (ˆD or ˆZ) is encountered by the function `scanf()` and it returns a value `EOF`. Remember, only the value of `flag`, NOT that of `n`, gets the value, `EOF`. The value of `n` will remain unchanged from its previous value when `scanf()` encounters end of file. Finally, the largest absolute value is printed out.

We have seen that `scanf()` returns a value of items read or `EOF`. It also performs the task of reading one or more items, converting them to internal form, and storing them at specified addresses. This additional task does not directly contribute to the returned value and is called a **side effect**. Functions may be used solely for their side effects, solely for their returned values, or for both side effects and returned values. For example, we use `printf()` for its side effect and ignore its value. We also frequently ignore the value of `scanf()`. In this section, we have used `scanf()` for both its side effect as well as its return value.

3.4.2 Standard Files and Redirection

As we stated, normally the standard input and standard output files are defined by default to be the keyboard and the screen. This may not always be convenient. For example, in our nice day program, we might want to gather statistics for an entire year of temperature data, or an entire decade. While we may have all this data readily available in a file, to use our program we would have to type it all in at the keyboard again (and what happens if we make a mistake and have to start all over). Operating systems such as Unix and MS-DOS allow a user to *redirect* the standard input and output files to files other than the keyboard and screen.

If our program in file, `niceday.c` were compiled using the command:

```
cc -o niceday niceday.c
```

producing the executable file `niceday`, we can execute the program with input data from a file called `temperatures` by typing the following command to the shell:

```
niceday < temperatures
```

The symbol < in the command redirects the standard input to come from the file `temperatures` instead of the keyboard.

Similarly, we can redirect the standard input to our payroll program, `pay5`, from a file containing monthly data for many employees:

```
pay5 < pay_data.march
```

However, in this case, unless we can read *very* fast, most of the output generated by the program will scroll past the screen before we can read it. In addition, we might want to save the results of our program execution in a file to send to a printer for a hard copy. A similar redirection of the standard output to a file can be done with the symbol > as follows:

```
pay5 < pay_data.march > pay_results.march
```

One problem remains with this technique: all output generated by the program from `printf()` statements will be redirected to the file, including the prompts we put in the program. In this case, the prompts are not necessary since the data is coming from a file, not from the user at the keyboard. For programs who's input and output are meant to be redirected from/to files, it is best to remove the `printf()` statements which produce prompts. We might even consider using conditional compilation to include or exclude the prompts, but remember, the program must be recompiled to change from one which prompts to one which does not, and vice versa.

3.5   Debugging Guidelines

As programs become large, finding bugs and debugging become a time consuming job. Debugging is an art that can be learned and developed. However, it requires plenty of experience in writing and debugging programs. The structured, top down approach to writing programs discussed in this chapter is one valuable tool for producing quality, working programs. However, there is no substitute for extensive programming experience and the best way to gain programming experience is to write, test, and debug programs; write, test, and debug programs; write, test, and debug programs; etc. etc.

Certain debugging guidelines are presented here to make the learning process easier:

1. The first step cannot be emphasized enough. Spend plenty of time in preparing the algorithm. A logically clear algorithm is much easier to debug than an ad hoc algorithm with many fixes for previously found bugs. Trial and error programming may never be bug free.

2. Use top down development for your algorithms, and use modular programming for your implementation. Top down development makes logic transparent at each stage and hides unnecessary details by relegating them to later stages. Modular programming localizes errors in small functions, which can be easily debugged.

3. Document your program using comments as you write it. It is a poor habit to delay documenting a program until it is done. Frequently, the very process of documenting a program makes the logic clearer and may well eliminate sources of errors.

4. Trace your program flow manually. This means: examine what happens to values of key variables at key points in the program. Use judicious starting values for these variables. Particularly, check values of variables at critical points, such as loop beginnings and ends, function calls, and other key points in the program.

5. If your compiler comes with a symbolic debugger, learn to use it. The time spent to learn the use of a debugger makes debugging of most programs an easier task.

6. Otherwise, use trace statements in your program. That is, use statements to print out values of key variables at key positions in the program to help pin-point the program segment where the bug may be located. The program segment containing a bug can be narrowed until the exact one or two lines of code are pin-pointed. It is then easier to spot the error and correct it. Trace statements are also called debug statements.

7. Pin-point the functions which generate errors. Rewrite the functions if they are overly complex or long. Many times, it is easier to rewrite a function than to rectify poor logic.

8. In program development, initially we need debug statements. Later, once a program is debugged, the debug statements must be removed. C provides conditional compilation which was discussed above. One use of conditional compilation is to conditionally compile debug statements. Initially the program, including debug statements, is compiled. Later, when the program has been debugged, it can be compiled without compiling the debug statements. Debug statements need not be removed from the code.

3.6 Common Errors

This section contains a list of common errors made by programmers — things to watch out for in your programming.

1. The wrong value is tested for `EOF` instead of the returned value of `scanf()`:

```
flag = scanf("%d", &n);
while (n != EOF)            /* should be: while (flag != EOF) */
        . . .
```

The value read is stored at the address given by `&n`, i.e. it is stored in `n`. The statement `scanf("%d", &n)` evaluates to a returned value which is either the number of data items read or `EOF`. In the above case, if an integer data item is read, the value returned will be 1. If no data item is read, `scanf()` returns `EOF`. The value returned by `scanf()` is stored in the variable `flag`, NOT in `n`. Test `flag` for `EOF`, NOT `n`.

2. An attempt is made by a called function to access a variable defined in the calling function.

```
#include <stdio.h>
#define TRUE 1
main()
{    int x, square(int x);

     x = 3;
     square(x);                 /* x cannot be unchanged by square() */
```

```
        printf("x = %d\n", x);          /* prints: x = 3 */
    }


    int square(int x)          /* x is a new object, with initial value */
                               /* passed by an argument in the function call */
    {
        x = x * x;             /* new x is changed */
        return TRUE;   /* a value is returned as the value of square() */
    }
```

The variable `x` in `main()` is a different object from `x` in `square()`. The value of the local cell, `x`, is changed in `square()`, but that does not affect the cell `x` in `main()`. The cell, `x`, in `main()` will still have the value 3 after the function call to `square()`. If `main()` needs the squared value of `x`, then `square()` should return the squared value of `x`, NOT `TRUE`. This returned value should be saved in a local variable in `main()`. For example, if the `return` statement in `square()` is:

```
    return x;
```

then the returned value can be saved in `main()`:

```
    x = square(x);
```

3. A function is not declared with a prototype statement. Without a prototype, the compiler will not be able to check for consistency in usage of the function. When a function is declared, the compiler checks for a correct number of arguments in function calls and checks for correct types.

4. A default declaration of a function assumes an integer type function value. If the actual definition of that function returns a non-integer type, then the compiler will consider it an attempt to redeclare a function. The compiler will flag it as an error.

5. An erroneous keystroke is entered when an end of file is to be entered. For example, an attempt is made to enter 0 or -1 for an end of file. These values are not the end of file keystrokes; they represent the possible values returned by `scanf()` when an end of file keystroke (^D or ^Z) is encountered.

3.7   Summary

This chapter has presented a key concept in the design of good programs; namely, top down design. Beginning with the algorithm, complex programming tasks are divided into logical subtasks which themselves may be further divided. This structured design is a form of information hiding — hiding the details of an operation in its abstraction. We have described how these logical subtasks may be implemented using functions in C. A function is a block of code, which when given some information, performs some operations on the data and returns a value. To invoke (call) a function, use a statement with the form:

<function_name> ([<argument>[,<argument>...]])

where each argument may be an arbitrary expression. A function is defined by specifying a function_header and a function_body. A function header takes the form:

<function_name> ([<parameter>[,<parameter>...]])

and a function body is simply a block containing local variable declarations followed by executable statements to perform the task of the function.

We saw that the <parameter>'s in the function header are really just special forms of variable declarations; containing a type specifier and an identifier. They declare additional local variables within the function which are initialized to the values passed as arguments in the call. We also saw how declaration statements can initialize variables when a block is entered:

<type_specifier><var_name> [=<init_expr>] [,   <var_name> [=<init_expr>]...];

Remember, all local variables local to a function may be accessed ONLY within the body of the function, not by functions calling this function and not by functions called by this function.

The value returned by a function is specified in a **return** statement of the form:

return <expression>;

If the last statement of the function is reached without executing a **return** statement, the function returns with an unknown return value.

Next we discussed another form of information hiding using compiler directives processed by the C preprocessor. These included macros, with and without arguments, including header files, and conditional compilation.

#define <symbol_name> <substitution_string>

#include <filename>
#include "filename"

#ifdef <identifier>

(and other variations of the **#if** directive).

Finally, we described the relationship between I/O in C and files, including end of file and redirection of standard input and output files.

130 **CHAPTER 3. DESIGNING PROGRAMS TOP DOWN**

3.8 Exercises

1. What will the following code do?

   ```
   #define SQ(x) x * x;
   printf("%d\n", SQ(3));
   ```

2. What will the following code do?

   ```
   #define SQ(x) x * x;
   printf("%d\n", SQ(2+3));
   ```

3. What will be the output of the following code:

   ```
   #define DEBUG 0
   #define TWICEZ  z + z

   main()
   {    int z = 5;

        #ifdef DEBUG
             printf("%d\n", TWICEZ * 2);
        #endif
   }
   ```

4. Check the following program for errors, if any, and use a manual trace to verify the program averages two numbers.

   ```
   #include <stdio.h>
   main()
   {    float x, y, average;

        printf("Type two numbers: ");
        scanf("%f %f", &x, &y);
        calc_avg(x, y);
        printf("Average of %f and %f is %f\n", x, y, average);
   }

   calc_avg(float a, float b)
   {
        return a + b / 2;
   }
   ```

5. Check the following program for errors, if any, and manually trace its execution.

```
main()
{    float x, y, average;

     printf("Type numbers\n");
     scanf("%f", &x);
     while (x != EOF) {
          printf("Number read = %f\n", x);
          scanf("%f", &x);
     }
}
```

3.9 Problems

1. Write a function `float speed_mph(float distance, float time);` where distance traveled is specified in feet and time interval is in seconds. The function should return the speed in miles per hour. A mile is 5280 feet. Show a manual trace.

2. Write a program that prints out an integer and its square for all integers in the range from 7 through 17. Use a function to calculate the square of an integer. Show a manual trace.

3. Write a program to sum all input numbers until end of file. The program should keep a count of the numbers entered and compute an average of the input numbers. Show a manual trace for the first three numbers.

4. Write a function `float max(float n1, float n2);` that returns the greater of `n1` and `n2`. Write a function `float min(float n1, float n2);` that returns the lesser of `n1` and `n2`. Write a program that reads in numbers and uses the above functions to find the maximum and the minimum of all the numbers. The end of input occurs when zero is typed. Zero is a valid number for determining the maximum and the minimum. Use debug statements to ensure that the maximum and the minimum are updated correctly.

5. Write a program that generates a table of equivalent Celsius (C) and Fahrenheit (F) temperatures from 0 to 212 degrees F. The table entries should be at five degree (F) intervals. Use a function to convert degrees F to C. The conversion between the two is given by:

   ```
   C = (F - 32) * 5.0 / 9.0
   ```

6. Write a program that uses a function to determine if a given year is a leap year. A year is a leap year if it is divisible by 400; or if it is divisible by 4 and it is not divisible by 100.

7. Write a function `float sum_rec_n(int n);` which returns the sum of the reciprocals of integers from 1 through n. Write a program that reads positive integers until end of file. For each positive integer, `x`, read, it prints `sum_rec_n(x)`. Reciprocals must be `float` values. Use a cast operator to convert an integer to `float` before the reciprocal is calculated.

8. Modify the pay calculation program of Figure 3.2 so that a function `print_data()` prints out the input data as well as the pay. The function `print_data()` should return the number of items it writes to the output.

9. Assume that C does not provide a multiply operator. Write a function, `int multiply(int n1, int n2);` that multiplies two integers `n1` and `n2`, and returns their product. Write a driver to test the function.

10. Write a function, `int factors(int n)`, where `n` is a positive integer. The function prints the smallest integer factors of `n`, excluding 1 and itself. For example, if `n` is 120, then `factors(n)` will print 2, 2, 2, 3, 5. The function returns TRUE if `n` has no factors and FALSE otherwise.

11. Write a program that reads a positive integer and tests if it is a prime number by using `factors()` from Problem 10

12. Write a function `int gcd(int n, int m);` that returns the greatest common divisor (GCD) of non-negative integers `n` and `m`. A GCD may be obtained as follows: if `m` is zero, then GCD is `n`; otherwise, replace current `n` by the current `m` and replace current `m` by `(n % m)`. Repeat until `m` becomes zero and GCD is found.

13. Assume that C does not have a divide operator. Write a function `int_divide()` with two integer arguments that returns an integer quotient when the first argument is divided by the second argument.

14. Assume that C does not have a modulus operator. Write a function `modulus()` with two integer arguments that returns the remainder when the first argument is divided by the second.

15. Write a program that prints the accumulated value of an initial investment invested at a specified annual interest and compounded annually for a specified number of years. Annual compounding means that the entire annual interest is added at the end of a year to the invested amount. The new accumulated amount then earns interest, and so forth. If the accumulated amount at the start of a year is `acc_amount`, then at the end of one year the accumulated amount is:

```
acc_amount = acc_amount + acc_amount * annual_interest
```

Use a function that returns the accumulated value given the amount, interest, and years. The prototype is:

```
float calc_acc_amt(float acc_amount, float annual_interest, int years);
```

16. Modify the function in Problem 15 so that the interest may be compounded annually, monthly, or daily. Assume 365 days in the year. Hint: Use an argument to specify annual, monthly, or daily compounding of interest. If interest is not to be compounded annually, the annual interest must be converted to monthly (i.e., interest / 12) or daily interest (i.e., interest / 365). The interest must then be compounded each year, each month, or each day as the case may be.

17. Write a function that calculates the factorial of an integer `n`. Use a driver to test the function for values of `n` from 1 to 7. Factorial of a positive integer, `n`, is given by the product of positive integers from 1 through `n`. Use a variable that stores the value of the cumulative product. The cumulative product is multiplied by a new value of an integer each time a loop is executed:

```
cum_prod = cum_prod * i;
```

The initial value of the cumulative product should be 1 so the first multiple accumulates correctly.

18. Write a function, `float pos_power(float base, int exponent);` which returns the value of base raised to a positive exponent. For example, if base is 2.0 and exponent is 3, the function should return 8.0. If the exponent is negative, the function should return 0.

19. Write a function, `neg_power()`, which returns base raised to a negative exponent.

20. Modify the functions in Problems 18 and 19 to write a function `float power(float base, int exponent)`; which returns an exponent power of base, where exponent may be positive or negative. If the exponent is zero, it should return 1.

21. Write a function `int weight(int n)`; where n is a positive integer. The function returns the weight of the most significant digit, i.e., the highest power of ten which does not exceed n. For example, if n is 2345, `weight(n)` returns 1000. Assume n is less than 10000.

22. Write a function, `int sig_dig_value(int n)`; that returns the integer value of the most significant digit of a positive integer n less than 10000. For example, if n is 2345, `sig_dig_value(n)` returns integer 2.

23. Write a function, `int suppress_msd(int n)`; that returns an integer value of a positive integer after the most significant digit is removed. For example, if n is 2345, `suppress_msd(n)` returns 345.

24. Use Problems 22 and 23 to write a function, `print_dig_int(int n)`; that prints successive integer values of digits of a positive integer n. Each digit value is printed on a separate line. For example, if n is 2345, `print_dig_int(n)` prints 2 on one line, 3 on the next, 4 on the next, and 5 on the last line.

25. Write a function `print_dig_float(float x)`; that writes the value of each digit of a floating point number x. For example, if x is 2345.1234, then `print_dig_float(x)` will print integer values of digits 2, 3, 4, 5, 1, 2, 3, and 4 in succession.

26. Write a macro to evaluate the sum of the squares of two parameters. Make sure the macro can be called with any argument expressions. Write a program that reads two values and uses the above macro to print the sum of the squares.

Chapter 4

Processing Character Data

So far we have considered only numeric processing, i.e. processing of numeric data represented as integer and floating point types. Humans also use computers to manipulate data that is not numeric such as the symbols used to represent alphabetic letters, digits, punctuation marks, etc. These symbols have a standard meaning to us, and we use them to represent (English) text. In the computer, the symbols used to store and process text are called **characters** and C provides a data type, `char`, for these objects. In addition, communication between humans and computers is in the form of character symbols; i.e. all data typed at a keyboard and written on a screen is a sequence of character symbols. The functions `scanf()` and `printf()` perform the tasks of converting between the internal form that the machine understands and the external form that humans understand.

In this chapter, we will discuss character processing showing how characters are represented in computers and the operations provided to manipulate character data. We will develop programs to process text to change it from lower case to upper case, separate text into individual words, count words and lines in text, and so forth. In the process, we will present several new control constructs of the C language, describe user interfaces in programs, and discuss input/output of character data.

4.1   A New Data Type: `char`

The complete set of characters that can be recognized by the computer is called the **character set** of the machine. As with numbers, the representation in the computer of each character in the set is done by assigning a unique bit pattern to each character. The typical character set consists of the following types of characters:

```
    Alphabetic lower case: 'a',..., 'z'
    Alphabetic upper case: 'A',..., 'Z'
    Digit symbols         : '0',..., '9'
    Punctuation           : '.', ',', ';', etc.
```

```
        Character                 Meaning

          '\a'                    alert (bell)
          '\b'                    backspace
          '\f'                    form feed
          '\n'                    newline
          '\r'                    carriage return
          '\t'                    horizontal tab
          '\v'                    vertical tab
          '\\'                    backslash
          '\''                    single quote
          '\"'                    double quote
          '\?'                    question mark
```

Table 4.1: Escape Sequences

```
Space               :  ' '
Special symbols     :  '@', '#', '$', etc.
Control Characters  :  newline, tab, bell or beep, etc.
```

For example, a digit symbol is character type data, so when we type *234* at the keyboard, we are typing a sequence of character symbols: '2', followed by '3', followed by '4'. The function `scanf()` takes this sequence and converts it to the internal form of the equivalent number, 234. Similarly, all writing on the screen is a sequence of characters so `printf()` takes the internal form of the number and converts it to a sequence of characters which are written onto the screen.

In C programs, variables may be declared to hold a single character data item by using the keyword `char` as the type specifier in the declaration statment:

```
    char ch;
```

A character constant is written surrounded by single quotation marks, e.g. 'a', 'A', '$', '!', etc. Only *printable character* constants can be written in single quotes, not control characters, so writing of non-printable control character constants requires special handling. In C, the backslash character, \, is used as an *escape character* which signifies something special or different from the ordinary and is followed by one character to indicate the particular control character. We have already seen one such *control sequence* in our `printf()` statments; the newline character, '\n'. Other frequently used control character constants written with an escape sequence, include '\t' for tab, '\a' for bell, etc. Table 4.1 shows the escape sequences used in C. The newline, tab, and space characters are called **white space** characters, for obvious reasons.

Let us consider a simple task of reading characters typed at the keyboard and writing them to the screen. The task is to copy (or echo) the characters from the input to the output. We will continue this task until there is no more input, i.e. until the end of the input file.

```
/*    File: copy0.c
      Programmer:
      Date:
      This program reads a stream of characters, one character at
      a time, and echoes each to the output until EOF.
*/

#include <stdio.h>
main()
{     char ch;         /* declaration for a character object ch */
      int flag;             /* flag stores the number of items read by scanf() */

      printf("***Copy Program***\n\n");
      printf("Type text, terminate with EOF\n");
      flag = scanf("%c", &ch);          /* read the first char */
      while (flag != EOF) {             /* repeat while not EOF */
          printf("%c", ch);             /* print the last char read */
          flag = scanf("%c", &ch);      /* read the next char, update flag */
      }                                 /* flag is EOF, ch may be unchanged */
}
```

Figure 4.1: Code for copy0.c

TASK

COPY0: Write out each character as it is read until the end of input file.

   The algorithm can be stated simply as:

```
read the first character
while there are more characters to read
    write or print the previously read character;
    read the next character
```

The code for this program is shown in Figure 4.1.

   The keyword `char` declares a variable, `ch`, of character data type. We also declare an integer variable, `flag`, to save the value returned by `scanf()`. Recall, the value returned is either the number of items read by `scanf()` or the value `EOF` defined in `stdio.h`. (We do not need to know the actual value of `EOF` to use it).

   After the title is printed, a character is read by the statement:

```
        flag = scanf("%c", &ch);
```

The conversion specification for character type data is `%c`, so this `scanf()` reads a single character from the input. If it is not an end of file keystroke, the character read is stored into `ch`, and the value returned by `scanf()`, 1, is saved in `flag`. As long as the value of `flag` is not `EOF`, the loop is entered. The loop body first prints the value of `ch`, i.e. the last character read, and then, the assignment statement reads a new character and updates `flag`. The loop terminates when `flag` is `EOF`, i.e. when an end of file keystroke is detected. Remember, `scanf()` does not store the value, `EOF` into the object, `ch`. DO NOT TEST THE VALUE OF `ch` FOR `EOF`, TEST `flag`. A sample session is shown below:

```
    ***Copy Program***

    Type text, terminate with EOF
    Now is the time for all good men
    Now is the time for all good men
    To come to the aid of their country.
    To come to the aid of their country.
    ^D
```

The sample session shows that as entire lines of characters are entered; they are printed. Each character typed is not immediately printed, since no input is received by the program until a newline character is typed by the user; i.e. the same buffering we saw for numeric data entry. When a newline is typed, the entire sequence of characters, including the newline, is placed in the keyboard buffer and `scanf()` then reads input from the buffer, one character at a time, up to and including the newline. In our loop, each character read is then printed. When the buffer is exhausted, the next line is placed in the buffer and read, and so on. So, `scanf()` is behaving just as it did for numeric data; each call reads one data item, in this case a character (`%c`). One notable difference between reading numeric data and character data is that when `scanf()` reads a character, leading white space characters are read, one character at a time, not skipped over as it is when reading numeric data.

4.1.1   The ASCII Character Set

Character data is represented in a computer by using standardized numeric codes which have been developed. The most widely accepted code is called the **American Standard Code for Information Interchange (ASCII)**. The ASCII code associates an integer value for each symbol in the character set, such as letters, digits, punctuation marks, special characters, and control characters. Some implementations use other codes for representing characters, but we will use ASCII since it is the most widely used. The ASCII characters and their decimal code values are shown in Table 4.2. Of course, the internal machine representation of characters is in equivalent binary form.

ASCII value	Character	ASCII value	Character	ASCII value	Character
000	^@	043	+	086	V
001	^A	044	,	087	W
002	^B	045	-	088	X
003	^C	046	.	089	Y
004	^D	047	/	090	Z
005	^E	048	0	091	[
006	^F	049	1	092	\
007	^G	050	2	093	]
008	^H	051	3	094	^
009	^I	052	4	095	_
010	^J	053	5	096	`
011	^K	054	6	097	a
012	^L	055	7	098	b
013	^M	056	8	099	c
014	^N	057	9	100	d
015	^O	158	:	101	e
016	^P	059	;	102	f
017	^Q	060	<	103	g
018	^R	061	=	104	h
019	^S	062	>	105	i
020	^T	063	?	106	j
021	^U	064	@	107	k
022	^V	065	A	108	l
023	^W	066	B	109	m
024	^X	067	C	110	n
025	^Y	068	D	111	o
026	^Z	069	E	112	p
027	^[	070	F	113	q
028	^\	071	G	114	r
029	^]	072	H	115	s
030	^^	073	I	116	t
031	^-	074	J	117	u
032	[space]	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

Table 4.2: ASCII Table

The ASCII table has 128 characters, with values from 0 through 127. Thus, 7 bits are sufficient to represent a character in ASCII; however, most computers typically reserve 1 byte, (8 bits), for an ASCII character. One byte allows a numeric range from 0 through 255 which leaves room for growth in the size of the character set, or for a sign bit. Consequently, a character data type may optionally represent signed values; however, for now, we will assume that character data types are unsigned, i.e. positive integer values, in the range 0—127.

Looking at the table, note that the decimal values 0 through 31, and 127, represent non-printable control characters. All other characters can be printed by the computer, i.e. displayed on the screen or printed on printers, and are called **printable characters**. All printable characters and many control characters can be input to the computer by typing the corresponding keys on the keyboard. The *character* column shows the key(s) that must be pressed. Only a single key is pressed for a printable character; however, control characters need either special keys on the keyboard or require the CTRL key pressed together with another key. In the table, a control key is shown by the symbol ^. Thus, ^A is control-A, i.e. the CTRL key kept pressed while pressing the key, A.

Notice that the character 'A' has the code value of 65, 'B' has the value 66, and so on. The important feature is the fact that the ASCII values of letters 'A' through 'Z' are in a contiguous increasing numeric sequence. The values of the lower case letters 'a' through 'z' are also in a contiguous increasing sequence starting at the code value 97. Similarly, the digit symbol characters '0' through '9' are also in an increasing contiguous sequence starting at the code value 48. As we shall see, this feature of the ASCII code is quite useful.

It must be emphasized that a digit symbol is a *character* type. Digit characters have code values that differ from their numeric equivalents: the code value of '0' is 48, that of '1' is 49, that of '2' is 50, and so forth. The table shows that the character with code value 0 is a control character, ^@, called the **NULL character**. Do NOT confuse it with the digit symbol '0'. Remember, a digit *character* and the equivalent *number* have different representations.

Besides using single quotes, it is also possible to write character constants in terms of their ASCII values in a C program, using either their octal or their hexadecimal ASCII values. In writing character constants, the octal or hexadecimal value follows the escape character, \, as shown in Table 4.3. At most three octal digits or at most two hexadecimal digits are needed. Note, after the escape backslash, a leading zero should not be included in writing octal or hexadecimal numbers. The last example in Table 4.3, '\0', is called the **NULL character**, whose ASCII value is zero. Once again, this is NOT the same character as the printable digit character, '0', whose ASCII value is 48.

4.1.2   Operations on Characters

As we just saw, in C, characters have numeric values and, therefore, may be used in numeric expressions. It is the ASCII code value of a character that is used in these expressions. For example (referring to Table 4.2), the value of 'a' is 97, and that of 'A' is 65. So, the expression 'a' - 'A' is evaluated as $97 - 65$, which is 32. As we shall see, this ability to do arithmetic with

```
        Character Constants                        Meaning

        '\007', '\07', '\7'            character whose value is octal 7

        '\101'                         character whose octal value is 101, or
                                       whose decimal value is 65, i.e. 'A'

        '\xB'                          character with hex. value B, i.e.
                                       with decimal value 11.

        '\0'                           character whose value is zero;
                                       it is called the NULL character
```

Table 4.3: Escape sequences with Octal & Hexadecimal values

character data simplifies character processing. When a character variable or constant appears in an expression, it is replaced by its ASCII value of type integer. When a character cell is assigned an integer value, the value is interpreted to be an ASCII value. In other words, a character and its ASCII value are used interchangeably as required by the context. While a cast operator can be used, we do not need it to go from character type to integer type, and vice versa. Here are some expressions using character variables and constants.

```
    ch = 97;                 /* ch <--- ASCII value 97, i.e., 'a' */
    ch = '\141';             /* ch <--- 'a'; octal 141 is decimal 97 */
    ch = '\x61';             /* ch <--- 'a'; hexadecimal 61 is decimal 97 */
    ch = 'a';                /* ch <--- 'a' */

    ch = ch - 'a' + 'A';     /* ch <--- 'A' */

    ch = 'd';
    ch = ch - 'a' + 'A';     /* ch <--- 'D' */
    ch = ch - 'A' + 'a';     /* ch <--- 'd' */
```

The first group of four statements merely assigns lower case 'a' to ch in four different ways: the first assigns a decimal ASCII value, the second assigns a character in octal form, the third assigns a character in hexadecimal form, the fourth assigns a character in a printable symbolic form. All of these statements have exactly the same effect.

The next statement, after the first group, assigns the value of an expression to ch. The right hand side of the assignment is:

```
    ch - 'a' + 'A'
```

Since the value of ch is 'a' from the previous four statements, the above expression evaluates to the value of 'a' - 'a' + 'A', i.e. the value of 'A'. In other words, the right hand side expression converts lower case 'a' to its upper case version, 'A', which is then assigned to ch. Since the values of lower case letters are contiguous and increasing (as are those of upper case letters) 'a' is less than 'b', 'b' less than 'c', and so forth. Also, the offset value of each letter from the base of the alphabet is the same for lower case letters as it is for upper case letters. For example, 'd' - 'a' is the same as 'D' - 'A'. So, if ch is any lower case letter, then the expression

```
ch - 'a' + 'A'
```

results in the upper case version of ch. This is because the value of ch - 'a' is the offset of ch from the lower case base 'a'; adding that value to the upper case base 'A' results in the upper case version of ch. So for example, if ch is 'f' then the value of the above expression is 'F'. Similarly, if ch is an upper case letter, then the expression

```
ch - 'A' + 'a'
```

results in the lower case version of ch which may then be assigned to a variable.

Using this fact, the last group of three statements in the above set of statements first assigns a lower case letter 'd' to ch. Then the lower case value of ch is converted to its upper case version, and then back to lower case.

As we mentioned, all lower case and upper case letters have contiguous and increasing values. The same is true for digit characters. Such a contiguous ordering makes it easy to test if a given character, ch, is a lower case letter, an upper case letter, or a digit. For example, any lower case letter has a value that is greater than or equal that of 'a' AND less than or equal to that of 'z'. From this, we can write a C expression that is True if and only if ch is a lower case letter:

```
(ch >= 'a' && ch <= 'z')
```

Here is a code fragment that checks whether a character is a lower case letter, an upper case letter, a digit, etc.

```
if (ch >= 'a' && ch <= 'z')
    printf("%c is a lower case letter\n", ch);
else if (ch >= 'A' && ch <= 'Z')
    printf("%c is an upper case letter\n", ch);
else if (ch >= '0' && ch <= '9')
    printf("%c is a digit symbol\n", ch);
else
    printf("%c is neither a letter nor a digit\n");
```

Observe the multiway decision and branch: `if ... else if ... else if ... else`. Only one of the branches is executed. The first `if` expression checks if the value of `ch` is between the values of `'a'` and `'z'`, a lower case letter. Only if `ch` is not a lower case letter, does control proceed to the first `else if` part, which tests if `ch` is an upper case letter. Only if `ch` is not an upper case letter, does control proceed to the next `else if` part, which tests if `ch` is a digit. Finally, if `ch` is not a digit, the last `else` part is executed. Depending on the value of `ch`, only one of the paths is executed with its corresponding `printf()` statement.

Let us see how the expression

```
(ch >= 'a' && ch <= 'z')
```

is evaluated. First, the comparison `ch >= 'a'` is performed; then, `ch <= 'z'` is evaluated; finally, the results of the two sub-expressions are logically combined by the AND operator. Evaluation takes place in this order because the precedence of the binary relational operators (`>=`, `<=`, `==`, etc.) is higher than that of the binary logical operators (`&&`, `||`). We could have used parentheses for clarity, but the precedence rules ensure the expression is evaluated as desired.

One very common error is to write the above expression analogous to mathematical expressions:

```
('a' <= ch <= 'z')
```

This would not be found to be an error by the compiler, but the effect will not be as expected. In the above expression, since the precedence of the operators is the same, they will be evaluated from left to right according to their associativity. The result of `'a' <= ch` will be either True or False, i.e. 1 or 0, which will then be compared with `'z'`. The result will be True since 1 or 0 is always less than `'z'` (ASCII value 122). So the value of the above expression will always be True regardless of the value of `ch` — not what we would expect.

Let's write a program using all this information. Our next task is to read characters until end of file and to print each one with its ASCII value and what we will call the *attributes* of the character. The attributes are a character's category, such as a lower case or an upper case letter, a digit, a punctuation, a control character, or a special symbol.

Task

ATTR: For each character input, print out its category and ASCII value in decimal, octal, and hexadecimal forms.

The algorithm requires a multiway decision for each character read. A character can only be in one category, so each character read will lead to the execution of one of the paths in a multiway decision. Here is the algorithm.

```
read the first character
```

```
repeat as long as end of file is not reached
      if the character is a lower case letter
            print the various character representations, and
            print that it is a lower case letter
      else if it is an upper case letter
            print the various character representations, and
            print that it is an upper case letter
      else if it is a digit
            print the various character representations, and
            print that it is a digit
            etc..
      read the next character
```

Notice we have abstracted the printing of the various representations of the character (as a character and its ASCII value in decimal, octal and hex) into a single step in the algorithm: `print the various character representations`, and we perform the same step in every branch of the algorithm. This is a classic situation calling for the use of a function: abstract the details of an operation and use that abstraction in multiple places. The code implementing the above algorithm is shown in Figure 4.2. We have *declared* a function `print_reps()` which is passed a single character argument and expect it to print the various representations of the character. We have used the function in the driver without knowing how `print_reps()` will perform its task.

We must now write the function `print_reps()`. The character's value is its ASCII value. When the character value is printed as a character with conversion specification `%c`, the symbol is printed; when printed as a decimal integer with conversion specification `%d`, the ASCII value is printed in decimal form. Conversion specification `%o` prints an integer value in octal form, and `%x` prints an integer value in hexadecimal form. We simply need a `printf()` call with these four conversion specifiers to print the character four times. The code for `print_reps()` is shown in Figure 4.3. The function simply prints its parameter as a character, a decimal integer, an octal integer, and a hexadecimal integer.

Sample Session:

```
***Character Attributes***

Type text, terminate with EOF
Aloha, ^A!
A, ASCII value decimal 65, octal 101, hexadecimal 41:  an upper case letter
l, ASCII value decimal 108, octal 154, hexadecimal 6c:  a lower case letter
o, ASCII value decimal 111, octal 157, hexadecimal 6f:  a lower case letter
h, ASCII value decimal 104, octal 150, hexadecimal 68:  a lower case letter
a, ASCII value decimal 97, octal 141, hexadecimal 61:  a lower case letter
,, ASCII value decimal 44, octal 54, hexadecimal 2c:  a punctuation symbol
^A, ASCII value decimal 1, octal 1, hexadecimal 1:  a control character
!, ASCII value decimal 33, octal 41, hexadecimal 21:  a punctuation symbol
```

```
/*    File: attr.c
      This program reads characters until end of file. It prints the
      attributes of each character including the ASCII value.
*/
#include <stdio.h>
int print_reps( char ch );

main()
{     char ch;
      int flag;

      printf("***Character Attributes***\n\n");
      printf("Type text, terminate with EOF \n");
      flag = scanf("%c", &ch);             /* read the first char */
      while (flag != EOF) {
           if (ch >= 'a' && ch <= 'z') {       /* lower case letter? */
                print_reps(ch);
                printf("lower case letter\n");
           }
           else if (ch >= 'A' && ch <= 'Z') {       /* upper case letter? */
                print_reps(ch);
                printf("an upper case letter\n");
           }
           else if (ch >= '0' && ch <= '9') { /* digit character? */
                print_reps(ch);
                printf("a digit symbol\n");
           }
           else if (ch == '.' || ch == ',' || ch == ';' || ch == ':' ||
                     ch == '?' || ch == '!') {       /* punctuation? */
                print_reps(ch);
                printf("a punctuation symbol\n");
           }
           else if (ch == ' ') {                     /* space? */
                print_reps(ch);
                printf("a space character\n");
           }
           else if (ch < 32 || ch == 127) {          /* control character? */
                print_reps(ch);
                printf("a control character\n");
           }
           else {                            /* must be a special symbol */
                print_reps(ch);
                printf("a special symbol\n");
           }
           flag = scanf("%c", &ch);          /* read the next char */
      }    /* end of while loop */
}          /* end of program */
```

Figure 4.2: Code for ASCII Attributes

```
/*   File: attr.c  --- continued
*/

int print_reps( char ch)
{
    printf("%c, ASCII value decimal %d, octal %o, hexadecimal %x: ",
                    ch,ch,ch,ch);
}
```

Figure 4.3: Printing character representations

```
, ASCII value decimal 10, octal 12, hexadecimal a:  a control character
^D
```

The last line printed refers to the newline character. Remember, every character including the newline is placed in the keyboard buffer for reading and, while scanf() skips over leading white space when reading a numeric data item, it does not do so when reading a character.

Can we improve this program? The driver (main()) shows all the details of character testing, beyond the logic of what is being performed here, so it may not be very readable. Perhaps we should define a set of macros to hide the details of the character testing expressions. For example, we might write a macro:

```
#define IS_LOWER(ch)     ((ch) >= 'a' && (ch) <= 'z')
```

Then the first if test in main() would be coded as:

```
        if ( IS_LOWER(ch) )  {
        . . .
```

which directly expresses the logic of the program. The remaining expressions can be recoded using macros similarly and this is left as an exercise at the end of the chapter.

One other thought may occur to us to further improve the program. Can we make the function print_reps() a little more abstract and have it print the various representations as well as the category? To do this we would have to give additional information to our new function, which we will call print_category(). We need to tell print_category() the character to print as well as its category. To pass the category, we assign a unique code to each category and pass the appropriate code value to print_category(). To avoid using "magic numbers" we define the following macros:

```
#define  LOWER     0
#define  UPPER     1
```

```
#define  DIGIT     2
#define  PUNCT     3
#define  SPACE     4
#define  CONTROL   5
#define  SPECIAL   6
```

Placing these defines (together with the comparison macros) in a header file, category.h, we can now recode the program as shown in Figure 4.4. The code for print_category() is also shown. Looking at this code, it may seem inefficient in that we are testing the category twice; once in main() using the character, and again in print_category() using the encoded parameter. Later in this chapter we will see another way to code the test in print_category() which is more efficient and even more readable. The contents of the header file, category.h is left as an exercise. The program shown in Figure 4.4 will behave exactly the same as as the code in Figure 4.2 producing the same sample session shown earlier.

4.1.3   Character I/O Using getchar() and putchar()

We have already seen how to read and print characters using our usual I/O built in functions, scanf() and printf(); i.e. the %c conversion specifier. We have also included the header file stdio.h in all our programs, because it contains the definition for EOF, and declares prototypes for these *formatted* I/O routines. In addition, stdio.h contains two other useful *routines*, getchar() and putchar(), which are simpler to use than the formatted routines for character I/O. We use the term *routine* for getchar() and putchar() because they are actually macros defined in stdio.h which use more general functions available in the standard library. (Often routines that are macros are loosely referred to as functions since their use in a program can appear like a function call, so we will usually refer to getchar() and putchar() as functions).

The function getchar() reads a single character from the standard input and returns the character value as the value of the function, but to accommodate a possible negative value for EOF, the type of the value returned is int. (Recall, EOF may be either 0 or -1 depending on implementation). So we could use getchar() to read a character and assign the returned value to an integer variable:

```
    int c;

    c = getchar();
```

If, after executing this statement, c equals EOF, we have reached the end of the input file; otherwise, c is the ASCII value of the next character in the input stream.

While int type can be used to store the ASCII value of a character, programs can become confusing to read — we expect that the int data type is used for numeric integer data and that char data type is used for character data. The problem is that char type, depending on implementation, may or may not allow negative values. To resolve this, C allows us to explicitly

```
/*   File: attr2.c
     This program reads characters until end of file. It prints the
     attributes of each character including the ASCII value.
*/
#include <stdio.h>
#include "category.h"

main()
{    char ch;
     int flag;

     printf("***Character Attributes***\n\n");
     printf("Type text, terminate with EOF \n");
     flag = scanf("%c", &ch);            /* read the first char */

     while (flag != EOF) {
          if( IS_LOWER(ch) ) print_category(LOWER, ch);
          else if( IS_UPPER(ch) ) print_category(UPPER, ch);
          else if( IS_DIGIT(ch) ) print_category(DIGIT, ch);
          else if( IS_PUNCT(ch) ) print_category(PUNCT, ch);
          else if( IS_SPACE(ch) ) print_category(SPACE, ch);
          else if( IS_CONTROL(ch) ) print_category(CONTROL, ch);
          else print_category(SPECIAL, ch);

          flag = scanf("%c", &ch);             /* read the next char */
     }    /* end of while loop */
}         /* end of program */

int print_category( int cat, char ch)
{
     printf("%c, ASCII value decimal %d, octal %o, hexadecimal %x: ",
                   ch,ch,ch,ch);
     if(      cat == LOWER )   printf("lower case letter\n");
     else if( cat == UPPER )   printf("an upper case letter\n");
     else if( cat == DIGIT )   printf("a digit symbol\n");
     else if( cat == PUNCT )   printf("a punctuation symbol\n");
     else if( cat == SPACE )   printf("a space character\n");
     else if( cat == CONTROL ) printf("a control character\n");
     else printf("a special symbol\n");
}
```

Figure 4.4: Alternate code for attributes program

declare a `signed char` data type for a variable, which can store negative values as well as positive ASCII values:

```
signed char c;

c = getchar();
```

An explicit `signed char` variable ensures that a character is stored in a character type object while allowing a possible negative value for `EOF`. The keyword `signed` is called a **type qualifier**.

A similar routine for character output is `putchar()`, which outputs its argument as a character to the standard output. Thus,

```
putchar(c);
```

outputs the ASCII character whose value is in `c` to the standard output. The argument of `putchar()` is expected to be an integer; however, the variable `c` may be either `char` type or `int` type (ASCII value) since the value of a `char` type is really an integer ASCII value.

Since both `getchar()` and `putchar()` are macros defined in `stdio.h`, any program that uses these functions must include the `stdio.h` header file in the program. Let us rewrite our copy program using these new character I/O routines instead of using `scanf()` and `printf()`. The new code is shown in Figure 4.5. Characters are read until `getchar()` returns `EOF`. Each character read is printed using `putchar()`. Sample output is shown below.

```
***File Copy Program***

Type text, EOF to quit
```
This is a test.
```
This is a test.
```
Now is the time for all good men
```
Now is the time for all good men
```
to come to the aid of their country.
```
to come to the aid of their country.
^D
```

The sample output shown here is for keyboard input so the effects of buffering the input is clearly seen: a line must be typed and entered before the characters become available in the input buffer for access by the program and then echoed to the screen.

Using `getchar()` and `putchar()` are simpler for character I/O because they do not require a format string as do `scanf()` and `printf()`. Also, `scanf()` stores a data item in an object whose address is given by its argument, whereas `getchar()` returns the value of the character read as its value. Both `scanf()` and `getchar()` return `EOF` as their value when they read an end of file marker in an input file.

```
/* File: copychr.c
    Program copies standard input to standard output.
*/
#include <stdio.h>

main()
{    signed char c;


    printf("***File Copy Program***\n\n");
    printf("Type text, EOF to quit\n");
    c = getchar();

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

Figure 4.5: Using getchar() and putchar()

4.1.4   Strings vs Characters

Frequently, we have needed to write constants that are not single characters but are sequences of characters. A sequence of zero or more characters is called a **string of characters** or simply a **string**. We have already used strings as arguments in function calls to printf() and scanf(). In C, there is no primitive data type for strings; however, as a convenience, string constants (also called **string literals**) may be written directly into a program using double quotes. The double quotes are not part of a string constant; they are merely used to delimit (define the limits,) of the string constant. (To include a double quote as part of a string, escape the double quote with the \ character).

```
"This is a string constant."
"This string constant includes newline character.\n"
"This string constant includes \" double quotes."
```

Escape sequences may of course be included in string constants. A string constant may even contain zero characters, i.e. an empty string:

```
""
```

Such a string is also called a **null string**.

Two adjacent strings are concatenated at compile time. Thus,
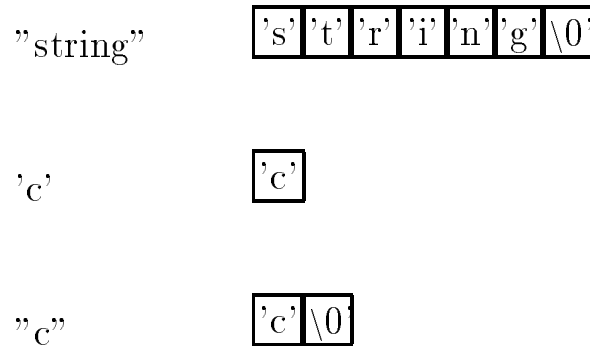
Figure 4.6: Strings

```
"John " "Doe"
```

are equivalent to:

```
"John Doe"
```

Whenever a string constant appears in a source program, the compiler stores the sequence of characters in contiguous memory locations and appends a `NULL` character to indicate the end of the string (see Figure 4.6). The compiler then replaces the string constant by the address where the characters are stored. Observe that a string of a single character is different from a character constant. Thus, `'c'` is a character constant; but, `"c"` is a string constant consisting of one character and the `NULL` character, as seen in the figure.

As we have said, a character constant takes on its ASCII value. The value of a string constant is the address where the string is stored. How this value can be used will be discussed in Chapter 6. For now, think of a string constant as a convenient representation, the exact nature of which will become clear later.

4.2 Sample Character Processing Functions

So far we have merely read and printed characters and determined their attributes. Character processing requires manipulation of input characters in meaningful ways. For example, we may wish to convert all lower case letters to upper case, all upper case letters to lower case, digit characters to their numeric equivalents, extract words, extract integers, and so forth. In this section we develop several programs which manipulate characters, beginning with simple example functions and continuing with programs for more complex text processing.

4.2.1   Converting Letter Characters

Our next task is to copy input characters to output as before except that all lower case letters are converted to upper case.

Task

COPY1: Copy input characters to output after converting lower case letters to upper case.

The algorithm is similar to COPY0, except that, before printing, each character it is converted to upper case, if necessary.

```
read the first character
repeat as long as NOT end of file
     convert character to upper case
     print the converted character
     read the next character
```

We will write a function, `uppercase()`, to convert a character. The function is given a character and if its argument is a lower case letter, `uppercase()` will return its upper case version; otherwise, it returns the argument character unchanged. The algorithm is:

```
if lower case convert to upper case,
otherwise, leave it unchanged;
```

The prototype for the function is:

```
char uppercase(char ch);
```

The code for the driver and the function is shown in Figure 4.7. The driver is straight forward; each character read is printed in its uppercase version. The while expression is:

```
((ch = getchar()) != EOF)
```

Here we have combined the operations of reading a character and testing for `EOF` into one expression. The innermost parentheses are evaluated first: `getchar()` reads a character and assigns the returned value to `ch`. The value of that expression, namely the value assigned to `ch`, is then compared with `EOF`. If it is not `EOF`, the loop executes, otherwise the loop terminates. The inner parentheses are essential. Without them, the expression is:

```
(ch = getchar() != EOF)
```

```
/*    File: copy1.c
      Programmer:
      Date:
      This program reads a stream of characters until end of file. Each
      character read is converted to its upper case version and printed
      out.
*/
#include <stdio.h>

#define IS_LOWER(c)     ((c) >= 'a' && (c) <= 'z')
#define TO_UPPER(c)     ((c) - 'a' + 'A')
char uppercase(char ch);

main()
{    signed char ch;

     printf("***Copy Program - Upper Case***\n\n");
     printf("Type text, terminate with EOF\n");

     while ((ch = getchar()) != EOF)
          putchar (uppercase(ch));  /* print value of uppercase(ch) */


}


/*    Function returns a lower case letter to an upper case. It returns
      all other characters unchanged.
*/
char uppercase(char c)
{
   if ( IS_LOWER(c) )        /* if c is a lower case letter */
      return TO_UPPER(c);            /* convert to upper case and return */
                                     /* otherwise, */
      return c;                      /* return c unchanged */
}
```

Figure 4.7: Code for upper case

Since the precedence of an assignment operator is the lowest, `getchar()` reads a character and the returned value is first compared to `EOF`:

```
getchar() != EOF
```

The value of this comparison expression, 0 or 1, is then assigned to `ch`: the wrong result is in `ch`. Of course, it is always best to use parentheses whenever there is the slightest doubt. Note, we have used the call to the function, `uppercase()`, as the argument for the routine, `putchar()`. The value returned from `uppercase()` is a character, which is then passed to `putchar()`.

The function, `uppercase()`, checks if `c` is a lower case letter (using the macro `IS_LOWER()`), in which case it returns the upper case version of `c`. We have used the macro `TO_UPPER()` for the expression to convert to upper case, making our program more readable. When the `return` statement is executed, control returns immediately to the calling function, thus, the code after the `return` statement is not executed. Therefore, in this case we do not need the `else` part of the `if` statement. In `uppercase()`, control progresses beyond the `if` statement only if `c` is not a lower case letter, where `uppercase()` returns `c` unchanged. A sample session is shown below:

```
***Copy Program - Upper Case***

Now is the time for all good men
NOW IS THE TIME FOR ALL GOOD MEN
To come to the aid of their country.
TO COME TO THE AID OF THEIR COUNTRY.
^D
```

4.2.2   Converting Digit Characters to Numbers

Next we discuss how digit symbols can be converted to their numeric equivalents and vice versa. As we have stated, the character `'0'` is not the integer, 0, `'1'` is not 1, etc. So it becomes necessary to convert digit characters to their numeric equivalent values, and vice versa. As we have seen, the digit values are contiguous and increasing; the value of `'0'` is 48, `'1'` is 49, and so forth. If we subtract the base value of `'0'`, i.e. 48, from the digit character, we can convert the digit character to its numeric equivalent; e.g. `'0'` - `'0'` is 0; `'1'` - `'0'` is 1; and so forth. Thus, if `ch` is a digit character, then its numeric equivalent is `ch` - `'0'`. Conversely, suppose `n` is a positive integer less than 10, (0, 1, 2, ..., 9). Then the corresponding digit character is `n` + `'0'`.

Using the sketch of an algorithm just described, we can write two functions that convert a digit character to its integer value, and an integer less than 10 to its character representation. These sound like operations that could be useful in a variety of programs, so we will put the functions in a file called `charutil.c`. These functions are the beginning of a library of character utility functions we will build. The code is shown in Figure 4.8. (We can also place the code for `uppercase()` from the previous example in this file as part ot the library). We have included the

```c
/* File: chrutil.c */
/*   This file contains various utility functions for processing characters  */

#include <stdio.h>
#include "chrutil.h"

/*   Function converts ch to an integer if it is a digit. Otherwise, it
     prints an error message.
*/
int dig_to_int(char ch)
{
    if (IS_DIGIT(ch))
        return ch - '0';
    printf("ERROR:dig_to_int:  %c is not a digit\n", ch);
    return ERROR;
}


/*   Function converts a positive integer less than 10 to a corresponding
     digit character.
*/
char int_to_dig(int n)
{
    if (n >= 0 && n < 10)
        return n + '0';
    printf("ERROR:int_to_dig:  %d is not in the range 0 to 9\n", n);
    return NULL;
}
```

Figure 4.8: Code for Character Utilities

```
/*   File:  charutil.h  */
/*   This file contains macros and prototypes for character utilities  */

#define   ERROR     -1

#define   IS_DIGIT(c)   ((c) >= '0' && (c) <= '9')
#define   IS_LOWER(c)   ((c) >= 'a' && (c) <= 'z')

int dig_to_int(char ch);
char int_to_dig(int n);
char uppercase(char ch);
```

Figure 4.9: Header file for Character Utilities

file `charutil.h` where the necessary macros and prototypes are located. This header file is shown in Figure 4.9.

The function `dig_to_int()` is given a character and returns an integer, namely the value of `ch - '0'` if `ch` is a digit character. Otherwise, it prints an error message and returns the value `ERROR`. Since valid integer values of digits are from 0 to 9, a value of -1 is not normally expected as a return value so we can use it to signify an error. (Note, we use a macro, in `charutil.h`, to define this "magic number"). In `int_to_dig()`, given an integer, `n`, the returned value is a digit character, `n + '0'`, if `n` is between 0 and 9; otherwise, a message is printed and the NULL (ASCII value 0) character is returned to indicate an error. We do not use `ERROR` in this case because `int_to_dig()` returns a `char` type value, which may not allow negative values. As was the case for the function `uppercase()` above, in these two functions, we have not used an `else` part. If the condition is satisfied, a `return` statement is executed. The control proceeds beyond the `if` part only if the condition is false. Returning some error value is a good practice when writing utility functions as it makes the functions more general and *robust*, i.e. able to handle valid and invalid data.

Let us consider the task of reading and converting a sequence of digit characters to an equivalent integer. We might add such an operation to our library of character utilities and call it `getint()` (analogous to `getchar()`). We will assume that the input will be a sequence of digit characters, possibly preceded by white space, but not by a plus or minus sign. Further, we will assume that the conversion process will stop when a character other than a digit is read. Usually, the delimiter will be white space, but any non-digit character will also be assumed to delimit the integer being read.

The function, `getint()`, needs no arguments and returns an integer. It will read one character at a time and accumulate the value of the integer. Let us see how a correct integer is accumulated in a variable, `n`. Suppose the digits entered are '3' followed by '4'. When we read the first digit, '3', and convert it to its integer value, we find that `n` is the number, 3. But we do not yet know if our integer is 3, or thirty something, or three hundred something, etc. So we read the next

character, and see that it is a digit character so we know our number is at least thirty something. The second digit is '4' which is converted to its integer value, 4. We cannot just add 4 to the previous value of n (3). Instead, we must add 4 to the previous value of 3 multiplied by 10 (the base — we are reading a decimal number). The new value of n is n * 10 + 4, or 34. Again, we do not know if the number being read is 34 or three hundred forty something, etc. If there were another digit entered, say '5', the new value of n is obtained by adding its contribution to the previous value of n times 10, i.e.

```
n * 10 + dig_to_int('5')
```

which is 345. Thus, if the character read, ch, is a digit character, then dig_to_int(ch) is added to the previously accumulated value of n multiplied by 10. The multiplication by 10 is required because the new digit read is the current rightmost digit with positional weight of 1; so the weight of all previous digits must be multiplied by the base, 10. For each new character, the new accumulated value is obtained by:

```
n = n * 10 + dig_to_int(ch);
```

We can write this as an algorithm as follows:

```
initialize n to zero
read the first character
repeat while the character read is a digit
    accumulate the new value of n by adding
        n * 10 + the integer value of the digit character
    read the next character
return the result
```

The code for getint() is shown in Figure 4.10. We have used conditional compilation to test our implementation by including debug statements to print the value of each digit, ch and the accumulated value of n at each step. The loop is executed as long as the character read is a digit. The macro, IS_DIGIT(), expands to an expression which evaluates to True if and only if its argument is a digit. Could we have combined the reading of the character and testing into one expression for the while?

```
while( IS_DIGIT(ch = getchar()))
```

The answer is NO! Recall, IS_DIGIT() is a macro defined as:

```
#define  IS_DIGIT(c)  ((c) >= '0' && (c) <= '9')
```

so IS_DIGIT(ch = getchar()) would expand to:

```
/*   File: charutil.c  - continued  */
/*   Function reads and converts a sequence of digit characters to an integer.  */

#define DEBUG

int getint()
{    int n;
     signed char ch;


     ch = getchar();                        /* read next char */
     while (IS_DIGIT(ch)) {                  /* repeat as long as ch is a digit */
         n = n * 10 + dig_to_int(ch);   /* accumulate value in n */
#ifdef DEBUG
printf("debug:getint: ch = %c\n", ch);   /* debug statement */
printf("debug:getint: n = %d\n", n);     /* debug statement */
#endif
         ch = getchar();                     /* read next char */
     }
     return n;                               /* return the result */

}
```

Figure 4.10: Code for `getint()`

```
((ch = getchar()) >= '0' && (ch = getchar()) <= '9')
```

While this is legal syntax (no compiler error would be generated), the function `getchar()` would
be called twice when this expression is evaluated. The first character read will be compared with
`'0'` and the second character read will be compared with `'9'` and be stored in the variable `ch`.
The lesson here is be careful how you use macros.

Notice we have used the function, `dig_to_int()` in the loop. This is an example of our modular
approach — we have already written a function to do the conversion, so we can just use it here,
trusting that it works correctly. What if `dig_to_int` ever returns the `ERROR` condition? In this
case, we know that that can never happen because if we are in the body of the loop, we know
that `ch` is a digit character from the loop condition. We are simply not making use of the full
generality of `dig_to_in()t`.

If, after adding the prototype for `getint()` to `charutil.h`:

```
int getint();
```

we compile the file `charutil.c`, we would get a load time error because there is no function `main()`
in the file. Remember, every C program must have a `main()`. To test our program, we can write
a short driver program which simply calls `getint()` and prints the result:

```
main()
{
    printf("***Test Digit Sequence to Integer***\n\n");
    printf("Type a sequence of digits\n");
    printf("Integer = %d\n", getint()); /* print value */
}
```

A sample session is shown below:

```
***Test Digit Sequence to Integer***

Type a sequence of digits
34
debug:getint:  ch = 3
debug:getint:  n = 16093
debug:getint:  ch = 4
debug:getint:  n = 160934
Integer = 160934
```

It is clear that something is wrong with the accumulated value of `n`. The first character '3' is read
correctly; but the value of `n` is 16093. The only possibility is that `n` does not have a correct initial
value; we have forgotten to initialize `n` to zero. A simple fix is to change the declaration of `n` in
`getint()` to:

```
int n = 0;
```

A revised sample session is shown below.

```
***Test Digit Sequence to Integer***

Type a sequence of digits
3456
debug:getint:  ch = 3
debug:getint:  n = 3
debug:getint:  ch = 4
debug:getint:  n = 34
debug:getint:  ch = 5
debug:getint:  n = 345
debug:getint:  ch = 6
debug:getint:  n = 3456
Integer = 3456
```

The trace shows that the program works correctly. The value of **n** is accumulating correctly. It is 3 after the first character, 34 after the next, 345, after the next, and 3456 after the last character. At this point, we should test the program with other inputs until we are satisfied with the test results for all the diverse inputs. If during our testing we enter the input:

```
***Test Digit Sequence to Integer***

Type a sequence of digits
        123
Integer = 0
```

we get the wrong result and no debug output. Notice, we have added some white space at the beginning of the line. In this case, the first character read is white space, not a digit. So the loop is never entered, no debug statements are executed, and the initial value of n, 0, is returned. We have forgotten to handle the case where the integer is preceded by white space. Returning to our algorithm, we must skip over white space characters after the first character is read:

```
initialize n to zero
read the first character
skip leading white space
repeat while the character read is a digit
    accumulate the new value of n by adding
        n * 10 + the integer value of the digit character
    read the next character
return the result
```

This added step can be implemented with a simple `while` loop:

```
while (IS_WHITE_SPACE(ch)) ch = getchar();
```

For readability, we have used a macro, `IS_WHITE_SPACE()`, to test `ch`. We can define the macro in `charutil.h` as follows:

```
#define   IS_WHITE_SPACE(c)   ((c) == ' ' || (c) == '\t' || (c) == '\n')
```

Compiling and testing the program again yields the correct result.

The program may now be considered debugged, it meets the specification given in the task, so we can eliminate the definition for DEBUG and recompile the program. However, at this point we might also consider the utility and generality of our `getint()` function. What happens if the user does not enter digit characters? What happens at end of the file? Only after the program is tested for the "normal" case, should we consider these "abnormal" cases. The first step is to see what the function, as it is currently written, does when it encounters unexpected input.

Let's look at `EOF` first. If the user types end of file, `getchar()` will return `EOF`, which is not white space and is not a digit. So neither loop will be executed and `getint()` will return the initialized value of `n`, namely 0. This may seem reasonable; however, a program using this function cannot tell the difference between the user typing zero and typing end of file. Perhaps we would like `getint()` to indicate end of file by returning `EOF` as `getchar()` does. This is easy to add to our program; before returning `n` we add a statement:

```
if(ch == EOF) return EOF;
```

Of course, if the implementation defines `EOF` as zero, nothing has changed in the behavior of the function. On the other hand, if the implementation defines `EOF` as -1, we can legally enter 0 as input to the program; however, should not expect -1 as a legal value. (In our implementation we have not allowed any negative number, so `EOF` is a good choice for a return value at end of file).

Next, let us consider what happens if the user types a non-digit character. If the illegal character occurs after some digits have been processed, e.g.:

32r

a manual trace reveals that the function will convert the number, 32, and return. If `getint()` is called again, the character, `'r'` will have been read from the buffer so the next integer typed by the user will be read and converted. (Note, this is different than what `scanf()` would do under these circumstances). This is reasonable behavior for `getint()`, so we need make no changes to our code.

If no digits have been typed before an illegal character, e.g.:

r 32

again, the character, `'r'` is not white space and not a digit, so `getint()` will return 0. As before, a program calling `getint()` cannot tell if the user entered zero or an error. It would be better if we return an error condition in this case, but if we return `ERROR`, defined in `charutil.h`, we may not be able to tell the difference between this error and `EOF`. The best solution to this problem is to change the definition of `ERROR` to be -2 instead of -1. This does not affect any other functions that have used `ERROR` (such as `dig_to_int()`) since they only need a unique value to return as an error condition. We can simply change the `#define` in `charutil.h` and recompile (see Figure 4.11). Finally, we must determine how to detect this error in `getint()`. As described above, we must know whether or not we have begun converting an integer when the error occurred. We can do this with a variable, called a **flag**, which stores the *state* of the program. We have called this flag `got_digit` (see Figure 4.12), and declare and initialize it to `FALSE` in `getint()`. If we ever execute the digit loop body, we can set `got_digit` to `TRUE`. Before returning, if `got_digit` is `FALSE` we should return `ERROR`, otherwise we return `n`.

All of these changes are shown in Figures 4.11 and 4.12. Notice we have included the header file, `tfdef.h` from before in the file `charutil.c` to include the definitions of `TRUE` and `FALSE`. We have also modified the test driver to read integers from the input until end of file. (Only the modified versions of `getint()` and the test driver,`main()` are shown in Figure 4.12. The functions `dig_to_int()` and `int_to_dig()` remain unchanged in the file).

Our `getint()` function is now more general and robust (i.e. can handle errors). Of particular note here is the method we used in developing this function. We started by writing the algorithm and code to handle the normal case for input. We then considered what would happen in the abnormal case, and made adjustments to the code to handle them only when necessary. This approach to program development is good for utilities and complex programs: get the normal and easy cases working first; then modify the algorithm and code for unusual and complex cases. Sometimes this approach requires us to rewrite entire functions to handle unusual cases, but often little or no extra code is needed for these cases.

4.2.3   Counting Words

The next task we will consider is counting words in an input text file (a file of characters). A word is a sequence of characters separated by delimiters, namely, white space or punctuation. The first word may or may not be preceded by a delimiter and we will assume the last word is terminated by a delimiter.

Task

CNT: Count the number of characters, words, and lines in the input stream until end of file.

Counting characters and lines is simple: a counter, `chrs`, can be incremented every time a character is read, and a counter, `lns`, can be incremented every time a newline character is read.

```
/* File: chrutil.h */
/*   This file contains various  macros and prototypes for character processing */

#define   ERROR      -2

#define   IS_DIGIT(c)   ((c) >= '0' && (c) <= '9')
#define   IS_LOWER(c)   ((c) >= 'a' && (c) <= 'z')
#define   IS_WHITE_SPACE(c)   ((c) == ' ' || (c) == '\t' || (c) == '\n')

int dig_to_int(char ch);
char int_to_dig(int n);
char uppercase(char ch);
int getint();
```

Figure 4.11: Revised Character Utility Header File

Counting words requires us to know when a word starts and when it ends as we read the sequence of characters. For example, consider the sequence:

```
    Lucky      luck
    ^     ^    ^    ^
```

We have shown the start and the end of a word by the symbol ˆ. There are several cases to consider:

- As long as no word has started yet AND the next character read is a delimiter, no new word has started.

- If no word has started AND the next character read is NOT a delimiter, then a new word has just started.

- If a word has started AND the next character is NOT a delimiter, then the word is continuing.

- If a word has started AND the character read is a delimiter, then a word has ended.

We can talk about the *state* of our text changing from "a word has not started" to "a word has started" and vice versa. We can use a variable, inword, as a flag to keep track of whether a word has started or not. It will be set to True if a word has started; otherwise, it will be set to False. If inword is False AND the character read is NOT a delimiter, then a word has started, and inword becomes True. If inword is True AND the new character read is a delimiter, then the word has ended and inword becomes False. With this information about the state, we can count a word either when it starts or when it ends. We choose the former, so each time the flag changes from False to True, we will increment the counter, wds. The algorithm is:

```
/* File: chrutil.c */
/*   This file contains various utility functions for processing characters  */

#include <stdio.h>
#include "tfdef.h"
#include "chrutil.h"

/*   Function reads the next integer from the input  */
int getint()
{    int n = 0;
     int got_dig = FALSE;
     signed char ch;


     ch = getchar();                      /* read next char              */
     while (IS_WHITE_SPACE(ch))           /* skip white space            */
             ch = getchar();
     while (IS_DIGIT(ch)) {               /* repeat as long as ch is a digit */
         n = n * 10 + dig_to_int(ch);     /* accumulate value in n       */
         got_dig = TRUE;
#ifdef DEBUG
printf("debug:getint: ch = %c\n", ch);   /* debug statement */
printf("debug:getint: n = %d\n", n);     /* debug statement */
#endif
         ch = getchar();                  /* read next char              */
     }
     if(ch == EOF) return EOF;            /* test for end of file        */
     if(!got_dig)  return ERROR;          /* test for no digits read     */
     return n;                            /* otherwise return the result */

}


/*   Dummy test driver for character utilities                */
/*   This driver will be removed after testing is complete  */
main()
{  int x;
     printf("***Test Digit Sequence to Integer***\n\n");
     printf("Type a sequence of digits\n");

     while((x = getint()) != EOF)
         printf("Integer = %d\n", x);        /* print value */
}
```

Figure 4.12: Revised Character Utility Code

```
    initialize all counters to zero, set inword to False
    while the character read, ch, is not EOF
        increment character count chrs
        if ch is a newline
            increment line count lns
        if NOT inword AND ch is NOT delimiter
            increment word count wds
            set inword to True
        else if inword and ch is delimiter
            set inword to False
    print results
```

We first count characters and newlines. After that, only changes in the state, `inword`, need to be considered; otherwise we ignore the character and read in the next one. Each time the flag changes from False to True, we count a word. We will use a function `delimitp()` that checks if a character is a delimiter, i.e. if it is a white space or a punctuation. (The name `delimitp` stands for "delimit predicate" because it tests is its argument is a delimiter and returns True or False). White space and punctuation, in turn, will be tested by other functions. The code for the driver is shown in Figure 4.13.

After printing the program title, the counts are initialized:

```
    lns = wds = chrs = 0;
```

Assignment operators associate from right to left so the rightmost operator is evaluated first; `chrs` is assigned 0, and the value of the assignment operation is 0. This value, 0, is then assigned to `wds`, and the value of that operation is 0. Finally, that value is assigned to `lns`, and the value of the whole expression is 0. Thus, the statement initializes all three variables to 0 as a concise way of writing three separate assignment statements.

The program driver follows the algorithm very closely. The function `delimitp()` is used to test if a character is a delimiter and is yet to be written. Otherwise, the program is identical to the algorithm. It counts every character, every newline, and every word each time the flag `inword` changes from False to True.

Source File Organization

We can add the source code for `delimitp()` to the source file `charutil.c` we have been building with character utility functions. In the last section we wrote a dummy driver in that file to test our utilities. Since we would like to use these utilities in many different programs, we should not have to keep copying a driver into this file. We will soon see how the code in `charutil.c` will be made a part of the above program without combining the two files into one (and without using the `#include` directive to include a code file). In our program file, `cnt.c`, we also include two header files besides `stdio.h`. These are: `tfdef.h` which defines symbolic constants TRUE and FALSE; and

```
/*    Program File: cnt.c
      Other Source Files: charutil.c
      Header Files: tfdef.h, charutil.h
      This program reads standard input characters and counts the number
      of lines, words, and characters. All characters are counted including
      the newline and other control characters, if any.
*/

#include <stdio.h>
#include "tfdef.h"
#include "charutil.h"

main()
{     signed char ch;
      int inword,          /* flag for in a word */
      lns, wds, chrs;      /* Counters for lines, words, chars. */

      printf("***Line, Word, Character Count Program***\n\n");
      printf("Type characters, EOF to quit\n");
      lns = wds = chrs = 0;    /* initialize counters to 0 */
      inword = FALSE;          /* set inword flag to False */

      while ((ch = getchar()) != EOF) {    /* repeat while not EOF */
          chrs = chrs + 1;                 /* increment chrs */
          if (ch == '\n')                  /* if newline char */
              lns = lns + 1;               /* increment lns */

          /* if not inword and not a delimiter  */
          if (!inword && !delimitp(ch)) { /* if not in word and not delim., */
              inword = TRUE;              /* set inword to True */
              wds = wds + 1;              /* increment wds */
          }

          else if (inword && delimitp(ch))  /* if in word and a delimiter*/
              inword = FALSE;               /* set inword to False */

      } /* end of while loop */
      printf("Lines = %d, Words = %d, Characters = %d\n",
              lns, wds, chrs);
} /* end of program */
```

Figure 4.13: Code for Count Words Driver

```
/* File: tfdef.h */
#define TRUE 1
#define FALSE 0

/*   File: charutil.h   - continued
     This file contains the prototype declarations for functions defined in
     charutil.c.
*/
int delimitp(char c);    /* Tests if c is a delimiter (white space, punct) */
int whitep(char c);      /* Tests if c is a white space */
int punctp(char c);      /* Tests if c is a punctuation */
```

Figure 4.14: Header Files for Word Count

charutil.h which declares prototypes for the functions defined in charutil.c and any related macros. Since we use these constants and functions in main(), we should include the header files at the head of our source file. Figure 4.14 shows the file tfdef.h and the additions to charutil.h.

The function delimitp() tests if a character is white space or punctuation. It uses two functions for its tests: whitep() which tests if a character is white space, and punctp() which tests if a character is punctuation. (We could have also implemented these as macros, but chose functions in this case). All these functions are added to the source file, charutil.c and are shown in Figure 4.15 This source file also includes tfdef.h, and charutil.h because the functions in the file use the symbolic constants TRUE and FALSE defined in tfdef.h and the prototypes for functions whitep() and punctp() declared in charutil.h are also needed in this file.

The source code for the functions is simple enough; delimitp() returns TRUE if the its parameter, c, is either white space or punctuation; whitep() returns TRUE if c is either a space, newline, or tab; and punctp() returns TRUE if c is one of the punctuation marks shown. All functions return FALSE if the primary test is not satisfied.

Our *entire* program is now contained in the two source files cnt.c and charutil.c which must be compiled separately and linked together to create an executable code file. Commands to do so are implementation dependent; but on Unix systems, the shell command line is:

```
cc -o cnt cnt.c charutil.c
```

The command will compile cnt.c to the object file, cnt.o, then compile charutil.c to the object file, charutil.o, and finally link the two object files as well as any standard library functions into an executable file, cnt as directed by the -o cnt option. (If -o option is omitted, the executable file will be called a.out). For other systems, the commands are generally similar; for example, compilers for many personal computers also provide an integrated environment which allows one to edit, compile, and run programs. In such an environment, the programmer may be asked to prepare a project file listing all source files. Once a project file is prepared and the project

```
/*   File: charutil.c  - continued */
#include "tfdef.h"
#include "charutil.h"
/*   Function returns TRUE if c is a delimiter, i.e., it is a white space
     or a punctuation. Otherwise, it returns FALSE.
*/
int delimitp(char c)
{
    if (whitep(c) || punctp(c))
         return TRUE;
    return FALSE;
}


/* Function returns TRUE if c is white space; returns FALSE otherwise. */
int whitep(char c)
{
    if (c == '\n' || c == '\t' || c == ' ')
         return TRUE;
    return FALSE;
}


/* Function returns TRUE if c is a punctuation; returns FALSE otherwise. */
int punctp(char c)
{
    if (c == '.' || c == ',' || c == ';' || c == ':'
            || c == '?' || c == '!')
         return TRUE;
    return FALSE;
}
```

Figure 4.15: Code for Word Count Utility Functions

option activated, a simple command compiles the source files, links them into an executable file, and executes the program. Consult your implementation manuals for details. This technique of splitting the source code for an entire program into multiple files is called **serarate compilation** and is a good practice as programs grow larger.

Once the above two files, `cnt.c` and `charutil.c` are compiled and linked, the resulting program may then be executed producing a sample session as shown below:

```
***Line, Word, Character Count Program***

Type characters, EOF to quit
Now is the time for all good men
To come to the aid of their country.
^D
Lines = 2, Words = 16, Characters = 70
```

Henceforth, we will assume separate compilation of source code whenever it is spread over more then one file. Since `main()` is the program driver, we will refer to the source file that contains `main()` as the **program file**. Other source files needed for a complete program will be listed in the comment at the head of the program file. In the comment, we will also list header files needed for the program. Refer to `cnt.c` in Figure 4.13 for an example of a listing which enumerates all the files needed to build or create an executable program. (The file `stdio.h` is not listed since it is assumed to be present in all source files).

Header files typically include groups of related symbolic constant definitions and/or prototype declarations. Source files typically contain definitions of functions used by one or more program files. We will organize our code so that a source file contains the code for a related set of functions, and a header file with the same name contains prototype declarations for these functions, e.g. `charutil.c` and `charutil.h`. As we add source code for new functions to the source files, corresponding prototypes will be assumed to be added in the corresponding header files.

Separate compilation has several advantages. Program development can take place in separate modules, and each module can be separately compiled, tested, and debugged. Once debugged, a compiled module need not be recompiled but merely linked with other separately compiled modules. If changes are made in one of the source modules, only that source module needs recompiling and linking with other already compiled modules. Furthermore, compiled modules of useful functions can be used and reused as building blocks to create new and diverse programs. In summary, separate compilation saves compilation time during program development, allows development of compiled modules of useful functions that may be used in many diverse programs, and makes debugging easier by allowing incremental program development.

4.2.4 Extracting Words

The final task in this section extends the word count program to print each word in the input stream of characters.

Task

WDS: Read characters until end of file and keep a count of characters, lines, and words. Also, print each word in the input on a separate line.

The logic is very similar to that of the previous program, except that a character is printed if it is in a word, i.e. if `inword` is True. We will decide whether to print a character only after a possible state change of `inword` has taken place. That way when `inword` changes from False to True (the first character of a word has been found) the character is printed. When `inword` changes from from True to False (a delimiter has been found ending the word) it is not printed, instead we print a newline because each word is to be printed on a new line. So our algorithm is:

```
    initialize counts to zero, set inword to False
    while the character read, ch, is not EOF
        increment character count, chrs
        if ch is a newline
            increment line count, lns
        if NOT inword AND ch is NOT delimiter
            increment word count, wds
            set inword to True
        else if inword and ch is delimiter
            set inword to False
            print a newline
        if inword
            print ch
    print results
```

and the code is shown in Figure 4.16. This code was generated by simply copying the file `cnt.c` and making the necessary changes as indicated in the algorithm. The program file is compiled and linked with `charutil.c`. and the following sample session is produced.

Sample Session:

```
***Word Program***

Type characters, EOF to quit
```
Now is the time for all good men
```
Now
is
the
time
for
all
good
men
```

```
/*    Program File: wds.c
      Other Source Files: charutil.c
      Header Files: tfdef.h, charutil.h
      This program reads standard input characters and prints each word on a
      separate line. It also counts the number of lines, words, and characters.
      All characters are counted including the newline and other control
      characters, if any.
*/

#include <stdio.h>
#include "tfdef.h"
#include "charutil.h"

main()
{     signed char ch;
      int inword,            /* flag for in a word                 */
      lns, wds, chrs;        /* Counters for lines, words, chars. */

      printf("***Line, Word, Character Count Program***\n\n");
      printf("Type characters, EOF to quit\n");
      lns = wds = chrs = 0;    /* initialize counters to 0 */
      inword = FALSE;          /* set inword flag to False */

      while ((ch = getchar()) != EOF) {   /* repeat while not EOF */
            chrs = chrs + 1;                  /* increment chrs       */
            if (ch == '\n')                   /* if newline char      */
                  lns = lns + 1;              /* increment lns        */

            /* if not inword and not a delimiter  */
            if (!inword && !delimitp(ch)) {   /* if not in word and not delim. */
                  inword = TRUE;                  /* set inword to True           */
                  wds = wds + 1;                  /* increment wds                */
            }

            else if (inword && delimitp(ch)) {  /* if in word and a delimiter*/
                  inword = FALSE;                 /* set inword to False       */
                  putchar('\n');                  /* end word with a newline   */
            }

            if (inword)                       /*  if in a word          */
                  putchar(ch);                /*  print the character  */

      } /* end of while loop */
      printf("Lines = %d, Words = %d, Characters = %d\n",
                  lns, wds, chrs);
} /* end of program */
```

Figure 4.16: Code fore extracting words

```
^D
Lines = 1, Words = 8, Characters = 33
```

In this section we have seen several sample programs for processing characters as well as some new programming techniques, in particular, splitting the source code for a program into files of related functions with separate compilation of each source code file. The executable program is then generated by linking the necessary object files. In the next section, we turn our attention to several new control constructs useful in character processing as well as in numeric programs.

4.3   New Control Constructs

Earlier in this chapter, we saw the use of a chain of if...else if constructs for a multiway decision. This is a common operation in programs so the C language provides an alternate multiway decision capability: the switch statement. In addition, two other control constructs are discussed in this section: the break and continue statements.

4.3.1   The switch Statement

In a switch statement, the value of an *integer valued expression* determines an alternate path to be executed. The syntax of the switch statement is:

switch (<expression>) <statement>

Typically, the <statement> is a compound statement with case labels.

switch (<expression>) {
      case $<e_1>$: $<stmt_1>$
      case $<e_2>$: $<stmt_2>$
      ...
      case $<e_{n-1}>$: $<stmt_{n-1}>$
      default: $<stmt_n>$
}

Each statement, except the last, starts with a case label which consists of the keyword case followed by a *constant expression*, followed by a colon. The constant expression, (whose value must be known at compile time) is called a **case expression**. An optional default label is also allowed after all the case labels. Executable statements appear after the labels as shown.

The semantics of the switch statement is as follows: The expression, <expression> is evaluated to an integer value, and control then passes to the first case label whose case expression value

matches the value of the switch expression. If no case expression value matches, control passes to the statement with the `default` label, if present. This control flow is shown in Figure 4.17. Labels play no role other than to serve as markers for transferring control to the appropriate statements. Once control passes to a labeled statement, the execution proceeds from that point and continues to process each of the subsequent statements until the end of the `switch` statement.

As an example, we use the `switch` statement to write a function that tests if a character is a vowel (the vowels are 'a', 'e', 'i', 'o', and 'u' in upper or lower case). If a character passed to this function, which we will call `vowelp()` (for vowel predicate), is one of the above vowels, the function returns True; otherwise, it returns False. We add the function to our file `charutil.c`, and the code is shown in Figure 4.18. If c matches any of the cases, control passes to the appropriate `case` label. For many of these cases, the <stmt> is empty, and the first non-empty statement is the `return TRUE` statement, which, when executed, immediately returns control to the calling function. If c is not a vowel, control passes to the `default` label, where the `return FALSE` statement is executed. While there is no particular advantage in doing so, the above function could be written with a `return` statement at every `case` label to return TRUE. The function `vowelp()` is much clearer and cleaner using the `switch` statement than it would have been using nested `if` statements or an `if` statement with a large, complex condition expression.

An Example: Encrypting Text

Remember, in a `switch` statement, control flow passes to the statement associated with the matching `case` label, and continues from there to all subsequent statements in the compound statement. Sometimes this is not the desired behavior. Consider the task of encrypting text in a very simple way, such as:

- Leave all characters except the letters unchanged.

- Encode each letter to be the next letter in a circular alphabet; i.e. 'a' follows 'z' and 'A' follows 'Z'.

We will use a function to print the next letter. The encrypt algorithm is simple enough:

```
read characters until end of file
    if a char is a letter
        print the next letter in the circular alphabet
    else
        print the character
```

Implementation is straight forward as shown in Figure 4.19. The program reads characters until end of file. Each character is tested to see if it is a letter using a function, `letterp()`. If it is a letter, `print_next()` is called to print the next character in the alphabet; otherwise, the character is printed as is. The function `letterp()` checks if a character passed as an argument is an alphabetic letter and returns True or False. The function is shown below and is added to our utility file, `charutil.c` (and its prototype is assumed to be added to the file `charutil.h`).
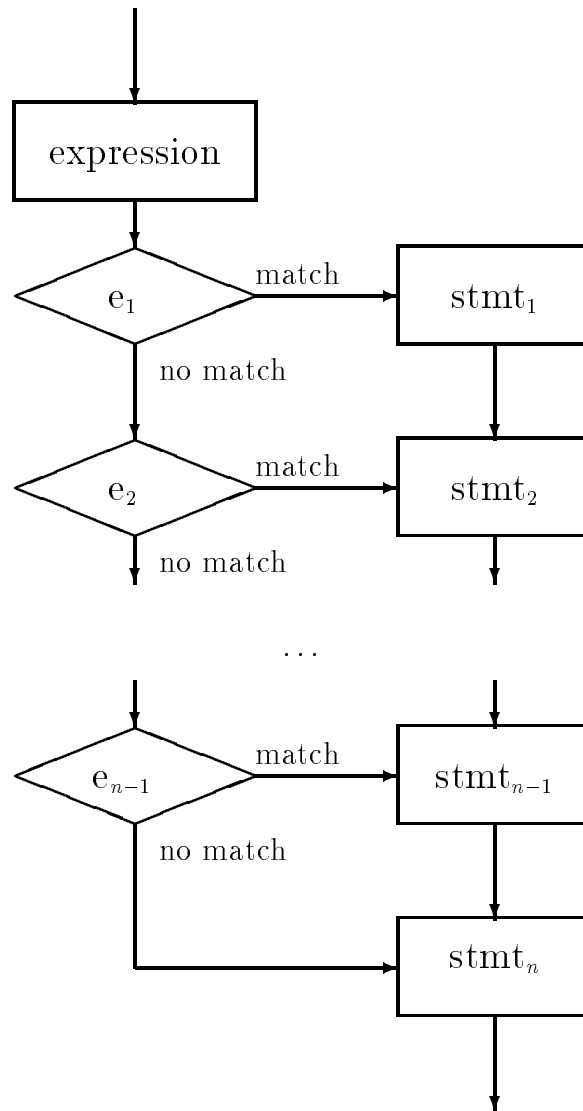
Figure 4.17: Control Flow for switch statement

```
/* File: charutil.c - continued */
/* File tfdef.h, which defines TRUE and FALSE, has already been
included in this file. */
/*   Function checks if c is a vowel. */
int vowelp(char c)
{
    switch(c) {
        case 'a':
        case 'A':
        case 'e':
        case 'E':
        case 'i':
        case 'I':
        case 'o':
        case 'O':
        case 'u':
        case 'U':   return TRUE;
        default:    return FALSE;
    }
}
```

Figure 4.18: Code for vowelp() Using a switch Statement

```
/*    File: encrypt.c
      Other Source Files: charutil.c
      Header Files: charutil.h
      This program encrypts text by converting each letter to the next letter
      in the alphabet. The last letter of the alphabet is changed to the first
      letter.
*/

#include <stdio.h>
#include "charutil.h"
void print_next(char c);

main()
{    signed char c;

     printf("***Text Encryption***\n\n");
     printf("Type text, EOF to quit\n");

     while ((c = getchar()) != EOF) {
          if (letterp(c))
                print_next(c);

          else
                putchar(c);
     }
}
```

Figure 4.19: Code for encrypt.c

```
/* File: charutil.c - continued */
/* Function tests if c is an alphabetic letter. */
int letterp(char c)
{
    if (IS_LOWER(c) || IS_UPPER(c))
        return TRUE;
    return FALSE;
}
```

It uses the macros IS_LOWER() and IS_UPPER(). We have already define IS_LOWER() in charutil.h; IS_UPPER() is similar:

```
#define   IS_UPPER(c)   ((c) >= 'A' && (c) <= 'Z')
```

and is added to charutil.h.

Let us consider the function, print_next(), which is passed a single alphabetic letter as an argument. It should print an altered letter, that is the next letter in a circular alphabet. The altered letter is the next letter in the alphabet, unless the argument is the last letter in the alphabet. If the argument is 'z' or 'Z', then the altered letter is the first letter of the alphabet, 'a' or 'A' respectively. There are two possible instances of the character c for which we must take special action, viz. when c is 'z' or c is 'Z'. The default case is any other letter, when the function should print c + 1, which is the ASCII value of the next letter.

We need a three way decision based on the value of a character c: is c the character 'z', or 'Z', or some other character? If it is 'z' print 'a'; else if it is 'Z' print 'A'; otherwise, print c + 1. We can easily implement this multiway decision using an if ... else ... construct.

```
    if (c == 'z')
        printf("%c", 'a');
    else if (c == 'Z')
        printf("%c", 'A');
    else
        printf("%c", c + 1);
```

Such multiway branches can also be implemented using the switch construct. Suppose we wrote:

```
    switch(c) {
        case 'z': printf("%c", 'a');
        case 'Z': printf("%c", 'A');
        default:  printf("%c", c + 1);
```

Will this do what we want? If c has the value 'z', the above switch statement would match the first case label and print 'a'. However, by the semantics of switch, it would then print 'A' followed by '{' (the character after 'z' in the ASCII table) — not what we want. Can we salvage this approach to multiway branching?

```
/*   File: encrypt.c - continued */
/*   Prints the next higher letter to c. Alphabet is assumed circular. */
void print_next(char c)
{
    switch(c) {
        case 'z': printf("%c", 'a');
                  break;
        case 'Z': printf("%c", 'A');
                  break;
        default:  printf("%c", c + 1);
    }
}
```

Figure 4.20: Implementing `print_next()` Using a `switch` Statement

4.3.2   The `break` Statement

C provides a statement for circumstances like this; the `break` statement. A `break` can only be used within a `switch` statement or any looping statement (so far we have only seen `while`). Its syntax is very simple:

```
break;
```

The semantics of `break` are to immediately terminate the closest enclosing compound statement; either the `switch` or the loop.

To fix our problem above, Figure 4.20 shows an implementation of `print_next()` using a `switch` statement. Once control passes to a label, the control continues down the line of statements until the `break` statement is encountered. In the above case, if c is 'Z', then an 'A' is printed and the switch statement is terminated. Similarly, if c is 'z', then 'a' is printed and the control passes to the next statement after the `switch`. If there is no match, then the control passes to the `default` label, and a character with value `c + 1` is printed. The `switch` statement ends at this point anyway, so no `break` is required.

Here is a sample session with the program after `encrypt.c` and `charutil.c` are compiled and linked.

```
***Text Encryption***

Type text, EOF to quit
this is a test
uijt jt b uftu
^D
```

```
/*  Function prints a character, its decimal, octal, and hex value
    and its category, using a switch statement
*/
int print_category( int cat, char ch)
{
    printf("%c, ASCII value decimal %d, octal %o, hexadecimal %x: ",
                ch,ch,ch,ch);
    switch(cat)   {
    case LOWER:   printf("lower case letter\n");
                  break;
    case UPPER:   printf("an upper case letter\n");
                  break;
    case DIGIT:   printf("a digit symbol\n");
                  break;
    case PUNCT:   printf("a punctuation symbol\n");
                  break;
    case SPACE:   printf("a space character\n");
                  break;
    case CONTROL: printf("a control character\n");
                  break;
    default:      printf("a special symbol\n");
    }
}
```

Figure 4.21: New Implementation of `print_category` using `switch`

This use of the `switch` statement with `break` statements in the various cases is a common and efficient way to implement a multiway branch in C. For example, we can now reimplement our `print_category()` function from Figure 4.4 as shown in Figure 4.21.

As mentioned above, the `break` statement can also be used to terminate a loop. Let us consider our previous word extraction task: reading text input and printing each word in the text (see Figure 4.16). However, now we will consider non-printable characters other than white space and the end of file marker as invalid. They will represent an error in the input and we will use a `break` statement to abort the program.

For this task, we will no longer count characters, words, and lines, simply extract words and print them, one per line. In our previous algorithm, each iteration of the loop processed one character and we used a flag variable, `inword` to carry information from one iteration to the next. For this program we will modify our algorithm so that each iteration of the loop will process one word. Each word is found by first skipping over leading delimiters, then, as long as we read printable, non-delimiter characters, we can print the word. The character terminating the word must be a delimiter unless it is a non-printable character or we have reached the end of file. In either of those cases, we abort the program, printing a message if a non-printable character was encountered. Otherwise, we print the newline terminating the word and process the next word.

Here is the revised algorithm with the code shown in Figure 4.22.

```
while there are more characters to read
     skip over leading delimiters (white space)
     while character is legal for a word
          print character
          read next character
     if EOF, terminate the program
     if character is non-printable,
          print a message and abort the program
     print a newline ending the word
```

The program uses two functions: `delimitp()` tests if the argument is a delimiter, and `illegal()` tests if the argument is not a legal character (printable or a delimiter). They are in the source file `charutil.c`; their prototypes are in `charutil.h`. We have already defined `delimitp()` (see Figure 4.15). We will soon write `illegal()`.

In the main loop, we skip over leading delimiters with a `while` loop, and then, as long legal "word" characters are read we print and read characters. If either of these loops terminates with **EOF**, the loop is terminated by a `break` statement and the program ends. (Note, if **EOF** is detected while skipping delimiters, the word processing loop will be executed zero times). If a non-printable, non-delimiter character is found, the program is aborted after a message is printed to that effect. Otherwise, the word is ended with a newline and the loop repeats.

Function `illegal()` is easy to write: legal characters are printable (in the ASCII range 32 through 126) or white space. Here is the function and its prototype.

```
/*   File: charutil.c - continued
     Header Files: tfdef.h, charutil.h
*/
/* Function tests if c is printable. */
int illegal(char c)
{
   if (IS_PRINT(c) || IS_WHITE_SPACE(c))
     return FALSE;
   return TRUE;
}

/*   File: charutil.h - continued */

#define  IS_PRINT(c)      ((c) >= 32 && (c) < 127)

int illegal(char c);      /* Tests if c is legal. */
```

We have also added the macro `IS_PRINT` to the header file. The program file `words.c` and the source file `charutil.c` can now be compiled and linked. A sample session when the program is

```
/*   File: words.c
     Other Source Files: charutil.c
     Header Files: tfdef.h, charutil.h
     This program reads text and extracts words until end of file. Only
     printable characters are allowed in a word. Upon encountering a control
     character, a message is printed and the program is aborted.
*/
#include <stdio.h>
#include "tfdef.h"
#include "charutil.h"  /* includes prototypes for delimitp(), printp() */

main()
{    signed char ch;

     printf("***Words: Non-Printable Character Aborts***\n\n");
     printf("Type text, EOF to quit\n");

     while ((ch = getchar()) != EOF) {  /* while characters remain to be read */

          while (delimitp(ch))          /* skip over leading delimiters      */
               ch = getchar();

          while (!delimitp(ch) && printp(ch)) {   /* process  a word     */
               putchar(ch);                        /* print ch            */
               ch = getchar();                     /* read the next char */
          }

          if (ch == EOF)                /* if end of file, terminate          */
               break;

          if (illegal(ch)) {/* if a control char, print msg and abort */
               printf("\nAborting - Control character present: ASCII %d\n",ch);
               break;
          }

          printf("\n");                 /* terminate word with newline        */
     }
}
```

Figure 4.22: Extracting Words Using `break`

executed is shown below.

```
***Words:  Non-Printable Character Aborts***

Type text, EOF to quit
Lucky you live HˆAwaiiˆA
Lucky
you
live
H
Aborting - Control character present:  ASCII 1
```

The message shows that the program is abnormally terminated due to the presence of a control character.

It is also possible, though not advisable, to use a `break` statement to terminate an otherwise infinite loop. Consider the program fragment:

```
n = 0;
while (1) {
    n = n + 1;
    if (n > 3) break;
    printf("Hello, hello, hello\n");
}
printf("Print statement after the loop\n");
```

The loop condition is the constant 1, which is always True so the loop body will be repeatedly executed, `n` will be incremented, and the message printed, until `n` reaches 4. The condition (`n > 3`) will now be True, and the `break` statement will be executed. This will terminate the `while` loop, and control passes to the print statement after the loop. If the `if` statement containing the `break` statement were not present, the loop would execute indefinitely.

While it is possible to use a `break` statement to terminate an infinite loop, it is not a good practice because use of infinite loops makes program logic hard to understand. In a well structured program, all code should be written so that program logic is clear at each stage of the program. For example, a loop should be written so that the normal loop terminating condition is immediately clear. Otherwise, program reading requires wading through the detailed code to see how and when the loop is terminated. A `break` statement should be used to terminate a loop only in cases of special or unexpected events.

4.3.3   The `continue` Statement

A `continue` statement also changes the normal flow of control in a loop. When a `continue` statement is executed in a loop, the current iteration of the loop body is aborted; however, control

transfers to the loop condition test and normal loop processing continues, namely either a new iteration or a termination of the loop occurs based on the loop condition. As might be expected, the syntax of the `continue` statement is;

```
continue;
```

and the semantics are that statements in the loop body following the execution of the `continue` statement are not executed. Instead, control immediately transfers to the testing of the loop condition.

As an example, suppose we wish to write a loop to print out integers from 0 to 9, except for 5. We could use the `continue` statement as follows:

```
n = 0;
while (n < 10) {
    if (n == 5) {
        n = n + 1;
        continue;
    }
    printf("Next allowed number is %d\n", n);
    n = n + 1;
}
```

The loop executes normally except when `n` is 5. In that case, the `if` condition is True; `n` is incremented, and the `continue` statement is executed where control passes to the testing of the loop condition, (`n < 10`). Loop execution continues normally from this point. Except for 5, all values from 0 through 9 will be printed.

We can modify our previous text encryption algorithm (Figure 4.19) to ignore illegal characters in its input. Recall, in that task we processed characters one at a time, encrypting letters and passing all other characters as read. In this case we might consider non-printable characters other than white space to be typing errors which should be ignored and omitted from the output.

The code for the revised program is shown in Figure 4.23. We have used the function, `illegal()`, from the previous program (it is in `charutil.c`) to detect illegal characters. When found, the `continue` statement will terminate the loop iteration, but continue processing the remaining characters in the input until `EOF`.

Sample Session:

```
***Text Encryption Ignoring Illegal Characters***

Type text, EOF to quit
Luck you live H^Awaii
Mvdl zpv mjjwf Ixbjj
```

```
/*   File: encrypt2.c
     Other Source Files: charutil.c
     Header Files: charutil.h
     This program encrypts text by converting each letter to the next letter
     in the alphabet. Illegal characters are ignored.
*/

#include <stdio.h>
#include "charutil.h"
void print_next(char c);

main()
{    signed char c;

     printf("***Text Encryption Ignoring Illegal Characters***\n\n");
     printf("Type text, EOF to quit\n");

     while ((c = getchar()) != EOF) {    /* while there are chars to process */

          if (illegal(c)) continue;          /* ignore illegal characters  */

          if (letterp(c))                     /* encrypt letters            */
               print_next(c);

          else
               putchar(c);                    /* print all others as is     */
     }
}
```

Figure 4.23: Code for Revised encrypt.c

```
/*    File: scan0.c
      This program shows problems with scanf() when wrong data is entered.
*/
#include <stdio.h>

main()
{    int cnt, n;

     printf("***Numeric and Character Data***\n\n");
     printf("Type integers, EOF to quit: ");

     cnt = 0;
     while ((scanf("%d", &n) != EOF) && (cnt < 4)) {
          printf("n = %d\n", n);

          cnt = cnt + 1;
          printf("Type an integer, EOF to quit: ");
     }
}
```

<p align="center">Figure 4.24: Code for Testing <code>scanf()</code></p>

$\hat{}D$

It should be noted that the use of `break` and `continue` statements is not strictly necessary. Proper structuring of the program, using appropriate loop and `if...else` constructs, can produce the same effect. The `break` and `continue` statements are best used for "unusual" conditions that would make program logic clearer.

4.4   Mixing Character and Numeric Input

We have seen how numeric data can be read with `scanf()` and character data with either `scanf()` or `getchar()`. Some difficulties can arise, however, when both numeric and character input is done within the same program. Several common errors in reading data can be corrected easily if the programmer understands exactly how data is read. In this section, we discuss problems in reading data and how they can be resolved.

The first problem occurs when `scanf()` attempts to read numeric data but the user enters the data incorrectly. (While the discussion applies to reading any numeric data, we will use integer data for our examples). Consider an example of a simple program that reads and prints integers as shown in Figure 4.24. In this program, `scanf()` reads an integer into the variable `n` (if possible) and returns a value which is compared with `EOF`. If `scanf()` has successfully read an integer,

the value returned is the number of conversions performed, namely 1, and the loop is executed. Otherwise, the value returned is expected to be EOF and the loop is terminated. The the first part of the while condition is:

```
(scanf("%d", &n) != EOF)
```

This expression both reads an item and compares the returned value with EOF, eliminating separate statements for initialization and update. The second part of the while condition ensures that the loop is executed at most 4 times. (The reason for this will become clear soon). The loop body prints the value read and keeps a count of the number of times the loop is executed. The program works fine as long as the user enters integers correctly. Here is a sample session that shows the problem when the user makes a typing error:

```
***Mistyped Numeric Data***

Type integers, EOF to quit:   23r
n = 23
Type an integer, EOF to quit:   n = 23
Type an integer, EOF to quit:   n = 23
Type an integer, EOF to quit:   n = 23
```

The user typed *23r*. These characters and the terminating newline go into the keyboard buffer, scanf() skips over any leading white space and reads characters that form an integer and converts them to the internal form for an integer. It stops reading when the first non-digit is encountered, in this case, the 'r'. It stores the integer value, 23, in n and returns the number of items read, i.e. 1. The first integer, 23, is read correctly and printed , followed by a prompt to type in the next integer.

   At this point, the program does not wait for the user to enter data; instead the loop repeatedly prints 23 and the prompt but does not read anything. The reason is that the next character in the keyboard buffer is still 'r'. This is not a digit character so it does not belong in an integer; therefore, scanf() is unable to read an integer. Instead, scanf() simply returns the number of items read as 0 each time. Since scanf() is trying to read an integer, it can not read and discard the 'r'. No more reading of integers is possible as long as 'r' is the next character in the buffer. If the value of the constant EOF is -1 (not 0), an infinite loop results. (That is why we have included the test of cnt to terminate the loop after 4 iterations).

   Let us see how we can make the program more tolerant of errors. One solution to this problem is to check the value returned by scanf() and make sure it is the expected value, i.e. 1 in our case. If it is not, break out of the loop. The while loop can be written as:

```
while ((flag = scanf("%d", &n)) != EOF) {
    if (flag != 1) break;
    printf("n = %d\n", n);
```

```
        printf("Type an integer, EOF to quit\n");
}
```

In the `while` expression, the inner parentheses are evaluated first. The value returned by `scanf()` is assigned to `flag` which is the value that is then compared to `EOF`. If the value of the expression is not `EOF`, the loop is executed; otherwise, the loop is terminated. In the loop, we check if a data item was read correctly, i.e. if `flag` is 1. If not, we break out of the loop. The inner parentheses in the while expression are important; the while expression without them would be:

```
(flag = scanf("%d", &n) != EOF)
```

Precedence of assignment operator is lower than that of the relational operator, ! =; so, the `scanf()` value is first compared with `EOF` and the result is True or False, i.e. 1 or 0. *This* value is then assigned to `flag`, NOT the value returned by `scanf()`.

The trouble with the above solution is that the program is aborted for a simple typing error. The next solution is to flush the buffer of all characters up to and including the first newline. A simple loop will take care of this:

```
while ((flag = scanf("%d", &n)) != EOF) {
    if (flag != 1)
        while (getchar() != '\n');
    else {
        printf("n = %d\n", n);
        printf("Type an integer, EOF to quit\n");
    }
}
```

If the value returned by `scanf()` when reading an integer is not 1, then the inner while loop is executed where, as long as a newline is not read, the condition is True and the body is executed. In this case, the loop body is an empty statement, so the condition will be tested again thus reading the next character. The loop continues until a newline is read. This is called **flushing the buffer**.

The trouble with this approach is that the user may have typed other useful data on the same line which will be flushed. The best solution is to flush only one character and try again. If unsuccessful, repeat the process until an item is read successfully. Figure 4.25 shows the revised program that will discard only those characters that do not belong in a numeric data item.

Sample Session:

```
***Mistyped Numeric Data:  Flush characters***

Type integers, EOF to quit
```

```
/*    File: scan1.c
      This program shows how to handle mistyped numeric data by flushing
      erroneous characters.
*/
#include <stdio.h>

#define DEBUG

main()
{     char ch;
      int flag, n;

      printf("***Mistyped Numeric Data: Flush characters***\n\n");
      printf("Type integers, EOF to quit\n");

      while ((flag = scanf("%d", &n)) != EOF) {

            if (flag != 1) {
                  ch = getchar();                /* flush one character */
#ifdef DEBUG
printf("debug:%c in input stream, discarding\n", ch);
#endif
            }

            else   printf("n = %d\n", n);

            printf("Type an integer, EOF to quit\n");
      }
}
```

Figure 4.25: Revised Code for Reading Integers

```
23rt         34
n = 23
Type an integer, EOF to quit
debug:r in input stream, discarding
Type an integer, EOF to quit
debug:t in input stream, discarding
Type an integer, EOF to quit
n = 34
Type an integer, EOF to quit
^D
```

The input contains several characters that do not belong in numeric data. Each of these is discarded in turn and another attempt is made to read an integer. If unable to read an integer, another character is discarded. This continues until it is possible to read an integer or the end of file is reached.

Even if the user types data as requested, other problems can occur with `scanf()`. The second problem occurs when an attempt is made to read a character after reading a numeric data item. Figure 4.26 shows an example which reads an integer and then asks the user if he/she wishes to continue. If the user types 'y', the next integer is read; otherwise, the loop is terminated. This program produces the following sample session:

```
***Numeric and Character Data***

Type an integer
23\n
n = 23
Do you wish to continue?  (Y/N): debug:
in input stream
```

The sample session shows that an integer input is read correctly and printed; the prompt to the user is then printed, but the program does not wait for the user to type the response. A newline is printed as the next character read, and the program terminates. The reason is that when the user types the integer followed by a RETURN, the digit characters followed by the terminating newline are placed in the keyboard buffer (we have shown the $\backslash n$ explicitly). The function `scanf()` reads the integer until it reaches the newline character, but leaves the newline in the buffer. This newline character is then read as the next input character into `c`. Its value is printed and the loop is terminated since the character read is not `'y'`.

A simple solution is to discard a single delimiting white space character after the numeric data is read. C provides a suppression conversion specifier that will read a data item of any type and discard it. Here are some examples:

```
scanf("%*c");          /* read and discard a character */
```

```
/*   File: mix0.c
     This program shows problems reading character data when it follows
     numeric data.
*/
#include <stdio.h>

#define  DEBUG

main()
{    char ch;
     int flag, n;

     printf("***Numeric and Character Data***\n\n");
     printf("Type an integer\n");

     while ((flag = scanf("%d", &n)) != EOF) {        /* continue until EOF */
          printf("n = %d\n", n);                      /* print n */

          printf("Do you wish to continue? (Y/N): ");  /* prompt */
          scanf("%c", &ch);                            /* read a character, */
#ifdef DEBUG
printf("debug:%c in input stream\n", ch);             /* type its value */
#endif

          if (ch == 'y')                               /* if char is 'y' */
               printf("Type an integer\n");            /* prompt */
          else                                         /* otherwise, */
               break;                                  /* terminate loop */
     }
}
```

Figure 4.26: Mixing Numeric and Character Data

```
scanf("%*d");                /* read and discard an integer */
scanf("%d%*c", &n);          /* read an integer and store it in n, */
                             /* then read and discard a character */
scanf("%*c%c", &ch);         /* read and discard a character, */
                             /* and read another, store it in ch, */
```

Figure 4.27 shows the revised program that discards one character after it reads an integer.

This program produces the following sample session:

```
***Numeric and Character Data***

Type an integer
23\n
n = 23
Do you wish to continue?  (Y/N): y\n
debug:y in input stream
Type an integer
34        \n
n = 34
Do you wish to continue?  (Y/N): debug:    in input stream
```

We have shown the terminating newline explicitly in the sample session input. The first integer is read and printed; one character is discarded and the next one read correctly as 'y' and the loop repeats. The next integer is typed followed by some white space and then a newline. The character after the integer is a space which is discarded and the following character is read. The new character read is another space, and the program is terminated because it is not a 'y'.

The solution is to flush the entire line of white space until a newline is reached. Then the next character should be the correct response. The revised program is shown in Figure 4.28 and the sample session is below:

```
***Numeric and Character Data***

Type an integer
23        \n
n = 23
Do you wish to continue?  (Y/N): y     \n
debug:y in input stream
Type an integer
34        \n
n = 34
Do you wish to continue?  (Y/N): n     \n
debug:n in input stream
```

```
/*    File: mix1.c
      This program shows how character data might be read correctly when it
      follows numeric data. It assumes only one white space character
      terminates numeric data. This character is suppressed.
*/
#include <stdio.h>

#define  DEBUG

main()
{    char ch;
     int flag, n;

     printf("***Numeric and Character Data***\n\n");
     printf("Type an integer\n");

     while ((flag = scanf("%d", &n)) != EOF) {
          printf("n = %d\n", n);

          printf("Do you wish to continue? (Y/N): ");
          scanf("%*c%c", &ch);      /* suppress a character, read another */
#ifdef DEBUG
printf("debug:%c in input stream\n", ch);
#endif

          if (ch == 'y')
               printf("Type an integer\n");
          else
               break;
     }
}
```

Figure 4.27: Revised Code for Mixing Data

```
/*   File: mix2.c
     This program shows how character data can be read correctly when it
     follows numeric data even if several white space characters follow
     numeric data.
*/
#include <stdio.h>

#define DEBUG

main()
{    char ch;
     int flag, n;

     printf("***Numeric and Character Data***\n\n");
     printf("Type an integer\n");

     while ((flag = scanf("%d", &n)) != EOF) {
          printf("n = %d\n", n);

          /* flush white space characters in a line; stop when newline read */
          while (getchar() != '\n');

          printf("Do you wish to continue? (Y/N): ");
          scanf("%c", &ch);
#ifdef DEBUG
printf("debug:%c in input stream\n", ch);
#endif

          if (ch == 'y')
               printf("Type an integer\n");
          else
               break;
     }
}
```

Figure 4.28: A Better Revision for Mixing Data

The first integer is read and printed, the keyboard buffer is flushed of all white space until the newline is read, and the next character is read to decide whether to continue or terminate the loop. The next character input is also terminated with white space; however, the next item to be read is a number and all leading white space will be skipped.

A final alternative might be to terminate the program only when the user types an 'n'; accepting any other character as a 'y'. This would be a little more forgiving of user errors in responding to the program. One should also be prepared for mistyping of numeric data as discussed above. A programmer should anticipate as many problems as possible, and should assume that a user may not be knowledgeable about things such as EOF keystrokes, will be apt to make mistakes, and will be easily frustrated with rigid programs.

4.5   Menu Driven Programs

Finally, we end this chapter by using what we have learned to improve the user interface to programs: we consider the case of a program driven by a *menu*. In a **menu driven program**, the user is given a set of choices of things to do (the menu) and then is asked to select a menu item. The driver then calls an appropriate function to perform the task selected by the menu item. A switch statement seems a natural one for handling the selection from the menu.

We will modify the simple version of our payroll program to make it menu driven. While a menu is not needed in this case, we use it to illustrate the concept. The menu items are: *get data*, *display data*, *modify data*, *calculate pay*, *print pay*, *help*, and *quit the program*. The user selects a menu item to execute a particular path; for example, new data is read only when the user selects the menu item, *get data*. On demand, the current data can be displayed so the user may make any desired changes. Pay is calculated only when the user is satisfied with the data.

Figure 4.29 shows the driver for this program. (The driver of any menu driven program will look similar to this). The program prints the menu and then reads a selection character. A switch is used to select the path desired by the user. The user may type a lower or an upper case letter; both cases are included by the case labels. Usually, the driver hides the details of processing individual selections, so we have implemented most selections as function calls. The only exception here is when the selection is *get data* where the actual statements to read the necessary data are included in the driver itself because to use a function, it would have to read several items and somehow return them. So far we only know how to write functions that return a single value. We will address this matter in Chapter 6.

Notice what happens if the user elects to quit the program: a standard library function, exit(), is called. This function is like a return statement, except that it terminates the entire program rather than return from a function. It may be passed a value which is returned to the *environment* in which the program runs. A value of 0 usually implies normal termination of a program; any other value implies abnormal termination.

After the appropriate function is called, we terminate the selected case with a break statement to end the switch statement . The control then passes to the statement after the switch state-

```
/*   File: menu.c
     An example of a menu driven program. The main() driver prints the menu,
     reads the selected item, and performs an appropriate task.  */
#include <stdio.h>
#include "payroll.h"

main()
{    signed char c;
     int id;
     float hours_worked, rate_of_pay, pay;

     printf("***Pay Calculation: Menu Driven***\n\n"); /* print title */
     print_menu();          /* Display the menu to the user            */
     while ((c = getchar()) != EOF) {        /* get user selection    */
         switch(c) {                   /* select an appropriate path   */
             case 'g':       /* should be a function get_data()     */
             case 'G': printf("Id number: ");
                       scanf("%d", &id);
                       printf("Type Hours worked and rate of pay\n");
                       scanf("%f %f", &hours_worked, &rate_of_pay);
                       break;
             case 'd':
             case 'D': display_data(id, hours_worked, rate_of_pay);
                       break;
             case 'm':
             case 'M': modify_data();
                       break;
             case 'c':
             case 'C': pay = calc_pay(hours_worked, rate_of_pay);
                       break;
             case 'p':
             case 'P': display_data(id, hours_worked, rate_of_pay);
                       print_pay(pay);
                       break;
             case 'h':
             case 'H': print_menu();
                       break;
             case 'q':
             case 'Q': exit(0);
             default:  printf("Invalid selection\n");
                       print_menu();
         } /* end of switch */
         while ((c = getchar()) != '\n');    /* flush the buffer */
     } /* end of while loop */
} /* end of program */
```

Figure 4.29: Code for menu driven program

ment, namely flushing the buffer. Let us see what would happen if this flush were not present. The user selects an item by typing a character and must terminate the input with a newline. The keyboard buffer will retain all characters typed by the user, including the newline. So if the user types:

$d\backslash n$

(showing the newline explicitly), the program would read the character, `'d'`, select the appropriate case in the `switch` statement and execute the path which displays data. When the `break` ends the `switch`, control returns to the `while` expression which reads the next character in the buffer: the newline. Since newline is not one of the listed cases, the `switch` will choose the `default` case and print an error message to the user. Thus, flushing the keyboard buffer always obtains a new selection. In fact, even if the user typed more than a single character to select a menu item (such as an entire word), the buffer will be flushed of all remaining characters after the first.

As we have mentioned before, a large program should be developed incrementally, i.e. in small steps. The overall program logic consisting of major sub-tasks is designed first without the need to know the details of how these sub-tasks will be performed. Menu driven programs are particularly well suited for incremental development. Once the driver is written, "dummy" functions (sometimes called **stubs**) can be written for each task which may do nothing but print a debug message to the screen. Then each sub-task is implemented and tested one at a time. Only after some of the basic sub-tasks are implemented and tested, should others be implemented. At any given time during program development, many sub-task functions may not yet be implemented. For example, we may first implement only *get data*, *print data*, and *help* (*help* is easy to implement; it just prints the menu). Other sub-tasks may be delayed for later implementation. Figure 4.30 shows example implementations of the functions used in the above driver. These are in skeleton form and can be modified as needed without changing the program driver. It should be noted that the linker will require that *all* functions used in the driver be defined. The stubs satisfy the linker without having to write the complete function until later.

The use of a menu in this example is not very practical. It is merely for illustration of the technique. The menu is normally printed only once, so if the user forgets the menu items, he/she may ask for help, in which case the menu is printed again. Also, if the user types any erroneous character, the default case prints an appropriate message and prints the menu.

4.6   Common Errors

1. Errors in program logic: The program does not produce the expected results during testing. Use conditional compilation to introduce debug statements.

2. The value of `getchar()` is assigned to a `char` type. It should be assigned to a `signed char` type if it is to be checked for a possibly negative value of `EOF`.

3. The keyboard buffer is not flushed of erroneous or unnecessary characters as explained in Section 4.4.

```
/* File: payroll.c  */
/* Prints the menu. */
void print_menu(void)
{    /* print the menu */
     printf("Select:\n");
     printf("\tG(et Data\n");
     printf("\tD(isplay Data\n");
     printf("\tM(odify Data\n");
     printf("\tC(alculate Pay\n");
     printf("\tP(rint Pay\n");
     printf("\tH(elp\n");
     printf("\tQ(uit\n");
}


/* Displays input data, Id number, hours worked, and rate of pay. */
void display_data(int id, float hrs, float rate)
{
     printf("Id Number %d\n", id);
     printf("Hours worked %f\n", hrs);
     printf("Rate of pay %f\n", rate);
}


/* Calculates pay as hrs * rate */
/*  a very simple version of calc_pay.  Out previous implementation
    could be used here instead.
*/
float calc_pay(float hrs, float rate)
{
     return hrs * rate;
}


/* Modifies input data. */
void modify_data(void)
{
     printf("Modify Data not implemented yet\n");
}


/* Prints pay */
void print_pay(float pay)
{
     printf("Total pay = %f\n", pay);
}
```

Figure 4.30: Menu Driven Functions

4. Improper use of relational operators:

```
if ('a' <= ch <= 'z')    /* should be ('a' <= ch && ch <= 'z') */
        ...
```

The operators are evaluated left to right: `'a' <=` ch is either True or False, i.e. 1 or 0. This value is compared with `'z'` and the result is always True.

5. An attempt is made to read past the end of the input file. If the standard input is the keyboard, it may or may not be possible to read input once the end of file keystroke is pressed. If the standard input is redirected, it is NOT possible to read beyond the end of file.

6. A **break** statement is not used in a **switch** statement. When a case expression matches the switch expression, control passes to that case label and control flow continues until the end of the **switch** statement. The only way to terminate the flow is with a **break** statement. Here is an example:

```
char find_next(char c)
{    char next;

     switch(c) {
          case 'z': next = 'a';
          default: next = c + 1;
     }
     return next;
}
```

Suppose c is `'z'`. The variable **next** is assigned an `'a'` and control passes to the next statement which assigns `c + 1` to **next**. In fact, the function always returns `c + 1` no matter what c is.

7. Errors in defining macros. Define macros carefully with parentheses around macro formal parameters. If the actual argument in a macro call is an expression, it will be expanded correctly only if the macro is defined with parentheses around formal parameters.

8. A header file is not included in each of the source files that use the prototypes and/or macros defined in it.

9. Repeated inclusion of a header file in a source file. If the header file contains defines, there is no harm done. BUT, if the header file contains function prototypes, repeated inclusion is an attempt to redeclare functions, a compiler error.

10. Failure to set environment parameters, such as the standard include file directory, standard library directory, and so forth. Most systems may already have the environment properly set, but that may not be true in personal computers. If necessary, make sure the environment is set correctly. Also, make sure that the compile and link commands correctly specify all the source files.

4.7 Summary

In this chapter we have introduced a new data type, `char`, used to represent textual data in the computer. Characters are represented using a standard encoding, or assignment of a bit pattern to each character in the set. This encoding is called **ASCII** and includes representations of several classes of characters such as alphabetic characters (letters, both upper and lower case), digit characters, punctuation, space, other special symbols, and control characters. We have seen how character variables can be declared using the `char` keyword as the type specifier in a declaration statement, and how character constants are expressed in the program, namely by enclosing them in single quotes, e.g. `'a'`. The ASCII value of a character can be treated as an integer value, so we can do arithmetic operations using character variables and constants. For example, we have discussed how characters can be tested using relational operators to determine their class, how characters can be converted, for example from upper to lower case, or from a digit to its corresponding integer value.

We have also discussed character Input/Output using `scanf()` and `printf()` with the `%c` conversion specifier, or the `getchar()` and `putchar()` routines defined in `stdio.h`. We have used these routines and operations to write several example programs for processing characters and discussed the organization of program code into separate source files. This later technique allows us to develop our own libraries of utility functions which can be linked to various programs, further supporting our modular programming style.

In this chapter we have also introduced several new control constructs available in the C language. These include the `switch` statement:

```
switch ( <expression>) <statement>
```

where the <statement> is usually a compound statement with `case` labels.

```
switch (<expression>) {
        case <e_1 >: <stmt_1 >
        case <e_2 >: <stmt_2 >
        ...
        case <e_{n-1} >: <stmt_{n-1} >
        default: <stmt_n >
}
```

The semantics of this statement are that the <expression> is evaluated to an integer type value and the `case` labels are searched for the first label that matches this value. If no match is found, the optional `default` label is considered to match any value. Control flow transfers to the statement associated with this label and proceeds to successive statements in the switch body. We can control which statements are executed further by using `return` or `break` statements with the `switch` body.

The syntax of the `break` statement is simply:

```
break;
```

and it may be used only within `switch` or loop bodies with the semantics of immediately termi-
nating the execution of the body. In loops, the `break` statement is best used to terminate a loop
under unusual or error conditions. A similar control construct available for loops is the `continue`
statement:

```
continue;
```

which immediately terminates the current *iteration* of the loop but returns to the loop condition
test to determine if the loop body is to be executed again.

We have also discussed some of the difficulties that can be encountered when mixing numeric
and character data on input. These difficulties are due to the fact that numeric conversion specifiers
(`%d` or `%f`) are "tolerant" of white space, i.e. will skip leading white space in the input buffer to
find numeric characters to be read and converted, while character input (using `%c` or `getchar()`)
is not. For character input, the next character, whatever it is, is read. In addition, numeric
conversions will stop at the first non-numeric character detected in the input, leaving it in the
buffer. We have shown several ways of handling this behavior to make the input tolerant of user
errors in Section 4.4.

Finally, we used the features of the language discussed in this chapter to implement a common
style of user interface: menu driven programs. Such a style of program also facilitates good top
down, modular design in the coding and testing of our programs.

4.8 Exercises

1. What is the value of each of the following expressions:

   ```
   ch = 'd';
   ```

   ```
   (a)   ((ch >= 'a') && (ch <= 'z'))
   (b)   ((ch > 'A') && (ch < 'Z'))
   (c)   ((ch >= 'A') && (ch <= 'Z'))
   (d)   ch = ch -'a' + 'A';
   (e)   ch = ch - 'A' + 'a';
   ```

2. What will be the output of the following:

   ```
   char ch;
   int d;

   ch = 'd';
   d = 65;
   printf("ch = %c, value = %d\n", ch, ch);
   printf("d = %d, d = %c\n", d, d);
   ```

3. Write the header file `category.h` discussed in section 4.1.2. Write the macros IS_UPPER(), IS_DIGIT(), IS_PUNCT(), IS_SPACE(), IS_CONTROL().

4. Write a code fragment to test:

   - if a character is printable but not alphabetic
   - if a character is alphabetic but not above 'M' or 'm'
   - if a character is printable but not a digit

5. Write separate loops to print out the ASCII characters and their values in the ranges:

   ```
   'a' to 'z',
   'A' to 'Z',
   '0' to '9'.
   ```

6. Are these the same: 'a' and "a"? What is the difference between them?

7. What will be the output of the source code:

   ```
   #define SQ(x) ((x) * (x))
   #define CUBE(x)  ((x) * (x) * (x))
   #define DIGITP(c) ((c) >= '0' && (c) <= '9')

   char c = '3';
   ```

```
        if (DIGITP(c))
                printf("%d\n", CUBE(c - '0'));
        else
                printf("%d\n", SQ(c - '0'));
```

8. Find the errors in the following code that was written to read characters until end of file.

```
        char c;

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

9. What will be the output of the following program?

```
    #include <stdio.h>
    main()
    {    int n, sum;
         char ch;

         ch = 'Y';
         sum = 0;
         scanf("%d", &n);
         while (ch != 'N') {
                 sum = sum + n;
                 printf("More numbers? (Y/N) ");
                 scanf("%c", &ch);
                 scanf("%d", &n);
         }
    }
```

10. What happens if `scanf()` is in a loop to read integers and a letter is typed?

11. What happens if `scanf()` reads an integer and then attempts to read a character?

12. Use a switch statement to test if a digit symbol is an even digit symbol.

13. Write a single loop that reads and prints all integers as long as they are between 1 and 100 with the following restrictions: If an input integer is divisible by 7 terminate the loop with a **break** statement; if an input integer is divisible by 6, do not print it but continue the loop with a **continue** statement.

4.9   Problems

1. First use graph paper to plan out and then write a program that prints the following message centered within a box whose borders are made up of the character *.

   ```
   Happy New Year
   ```

2. Write a program to print a character corresponding to an ASCII value or vice versa, as specified by the user, until the user quits. If the character is not printable, print a message.

3. Write a function that takes one character argument and returns the following: if the argument is a letter, it returns the position of the letter in the alphabet; otherwise, it returns FAIL, whose value is -1. For example, if the argument is 'A', it returns 0; if the argument is 'd', it returns 3, and so forth. Define and use macros to test if a character is a lower case letter or an upper case letter.

4. Use a switch statement to write a function that returns TRUE if a character is a consonant and returns FALSE otherwise.

5. Use a switch statement to write a function that returns TRUE if a digit character represents an odd digit value. If the character is not an odd digit, the function returns FALSE.

6. Write a program to count the occurrence of a specified character in the input stream.

7. Write a program that reads in characters until end of file. The program should count and print the number of characters, printable characters, vowels, digits, and consonants in the input. Use functions to check whether a character is a vowel, a consonant, or a printable character. Define and use macros to test if a character is a digit or a letter.

8. Modify the program in Chapter 2 to find prime numbers so that the inner loop is terminated by a break statement when a number is found not to be prime.

9. Write a function that takes two arguments, replicate(int n, char c);, and prints the character, c, a number, n, times.

10. Use replicate() to write a function, drawrect(), that draws a rectangle of length, g, and width, w. The dimensions are in terms of character spaces. The rectangle top left corner is at top, t, and left, l. The arguments, g, w, t, and l are integers, where t and l determine the top left corner of the rectangle, and the length of the rectangle should be along the horizontal. Use '*' to draw your lines. Write a program that repeatedly draws rectangles until length and width specified by the user are both zero.

11. Repeat 10, but modify drawrect() to fillrect() that draws a rectangle filled in with a specified fill character.

12. Write a function that draws a horizontal line proportional to a specified integer between the values of 0 and 50. Use the function in a program to draw a bar chart, where the bars are horizontal and in proportion to a sequence of numbers read.

13. Write a function to encode text as follows:

   a. If the first character of a line is an upper case letter, then encode the first character to one that is 1 position higher in a circular alphabet. Move the rest of the characters in the line up by 1 position in a circular printable part of the ASCII character set.

   b. If the first character of a line is a lower case letter, then move the first character down by 2 positions in a circular alphabet. Move the rest of the characters in the line down by 2 positions in a circular printable part of the ASCII character set.

   c. If the first character of a line is white space, then terminate the input.

   d. Otherwise, if the first character of a line is not a letter, then move all characters in the line down by 1 position in a circular printable part of the ASCII character set.

14. Write a function to decode text that was encoded as per Problem 13.

15. Write a menu-driven program that combines Problems 13 and 14 to encode or decode text as required by the user. The input for encoding or decoding is terminated when the first character of a line is a space. The commands are: *encode*, *decode*, *help*, and *quit*.

16. Write a function that takes three arguments, two float numbers and one arithmetic operator character. It returns the result of applying the operator to the two numbers. Using the function, write a program that repeatedly reads a float number, followed by an arithmetic operator, followed by a float number; each time it prints out the result of applying the operator to the numbers.

17. Modify the program in Problem 16 to allow further inputs of a sequence of an operator followed by a number. Each new operator is to be applied to the result from the previous operation and the new number entered. The input is terminated by a newline. Print only the final result.

18. Read and convert a sequence of digits to its equivalent integer. Any leading white space should be skipped. The conversion should include digit characters until a non-digit character is encountered. Modify the program so it can read and convert a sequence of digit characters preceded by a sign, + or -.

19. Write a program that converts the input sequence of digit characters, possibly followed by a decimal point, followed by a sequence of digits, to a float number. The leading white space is skipped and the input is terminated when a character not admissible in a float number is encountered.

20. Modify the above program to include a possible leading sign character.

21. Write a function that takes a possibly signed integer as an argument, and converts it to a sequence of characters.

22. Write a program that takes a possibly signed floating point number and converts it to a sequence of characters with 4 digits after the decimal point.

23. Modify the word extraction program, `wds.c`, in Figure 4.16. It should count words with exactly four characters and words with five characters. Assume the input consists of only valid characters and white space.

24. Write a program that reads in characters until end of file. The program should identify each *token*, i.e. a word after skipping white space. The only valid token types are: *integer* and *invalid*. White space delimits words but is otherwise ignored. An integer token is a word that starts with a digit and is followed by digits and terminates when a non-digit character is encountered. An invalid token is made up of any other single character that does not belong to an integer. Print each token as it is encountered as well as its type. Here is a sample session:

```
Type text, EOF to quit:   3456 a23b
3456 integer
a invalid
23 integer
b invalid
Type text, EOF to quit:   ^D
```

25. Modify the program in Problem 24 so it also allows an identifier as a valid token. An identifier starts with a letter and may be followed by a sequence of letters and/or digits.

26. Modify the program in Problem 25 so that tokens representing float numbers are also allowed. A float token must start with a digit, may be followed by a sequence of digits, followed by a decimal point, followed by zero or more digits. Here is a sample session:

```
Type text, EOF to quit:   The ID Number is 123, not 123.
The Identifier
ID Identifier
Number Identifier
is Identifier
123 Integer
, Invalid
not Identifier
123. Float

Type text, EOF to quit:   pay = 1.5 * hours * rate;
pay Identifier
= Invalid
1.5 Float
* Invalid
hours Identifier
* Invalid
rate Identifier
; Invalid
Type text, EOF to quit:   ^D
```

Hint: Skip leading delimiters; test the first non-delimiter, and build a word of the appropriate type. An integer and a float are distinguished by the presence of a decimal point.

Chapter 5

Numeric Data Types and Expression Evaluation

In the preceding chapters we have introduced all the basic tools needed to write programs in C: the control constructs and operators of the language, as well as the basic data types for integer, floating point, and character data. Using these basic tools, we have been able to write programs for both numeric processing and non-numeric, character, processing.

In this chapter we will introduce several useful features of C that allow greater flexibility in program writing and allow a greater range of values and precision. We will first take a closer look at integer and floating point data types; their size, and limitations, and will introduce sub-types of integers, and double precision floating point numbers. We will formalize the order of evaluation of operators in expressions as well as the type of the expression value when several data types are are present as operands. We will also introduce several C statements that are possible alternatives for statements already discussed and describe some new operators.

5.1   Representing Numbers

As we saw in Chapter 1, the range of possible values of objects depends on the sizes used to represent them. The finite size of an object puts a limit on the range of values that can be stored in it. Integer objects have a limit on the range of positive and negative integers. Floating point numbers have limits on the number of significant digits (known as the **precision**) as well as on the range of the exponents (limiting the range of numbers). We will illustrate the reasons for these limits by analogy with decimal representation.

Let us represent integers using a finite number of decimal digits, say only five digits are allowed. We can use these digits to represent unsigned positive integers in the range 0 to 99999. If we wish to represent both positive and negative numbers, we need one digit to encode the sign, + or -, and can then use only the remaining four digits to represent the absolute value of an integer. So, with five digits, we can represent positive and negative integers in the range -9999 to +9999. If we had

more digits to represent integers, the range of values will be appropriately greater.

Now let us use the same five digits to represent floating point numbers in scientific notation, i.e. a fractional part multiplied by a power of ten. For our discussion, we will assume that the fractional part is less than 1 and that the exponent of ten can be positive or negative. For example:

```
.234E3
.987E-2
-.345E2
```

The numbers are shown as a fraction times some power of ten where the exponent is shown after the `E`. The first number is 234.0, the second is .00987, the third is -34.5.

When we represent numbers using this system, we do not need to store the decimal point (it is always in the same place) or the base (it is always `E`, standing for 10). So, of our 5 digits, let us say that we use three digits for the fractional part and two digits for the exponent. One digit of the fractional part and one digit of the exponent is reserved for the sign. This leaves only two digits for the absolute value of the fractional part, and it leaves one digit for the absolute value of the exponent. Thus, the range of values for the fractional part is $-.99$ to $+.99$ and the range for exponents is -9 to $+9$.

Even though the range of actual values is quite large (we can represent numbers from almost negative one billion to positive one billion), there are only two significant digits of precision; all other digits will be zeros contributed by the power of ten. So, the range of numbers is from $-990,000,000$ to $+990,000,000$ (`-.99E+9` to `+.99E+9`). With this scheme, it would be impossible to represent a number such as 123.4567 *exactly*. The best we can do is represent it as `+.12E+3`, which is the number 120 — not nearly as accurate as 123.4567. We have a loss of precision (or accuracy) because of the limited number of digits we have for representing floating point numbers. There is a slight distinction between *precisions* and *accuracy*. In the above representation scheme, we can always say there are 2 digits of precision; however, the accuracy depends on the value of the exponent. The smallest number we can represent is .00000000099 (`+.99E-9`), which is pretty darn accurate. However, if the exponent is $+9$, our accuracy is only ± 5 million. If more digits are used to represent floating point numbers, the precision and the range can be greater. For example, if 6 digits were allowed, with four digits for a signed fractional part, we could represent 123.4567 as `+.123E3`, which is 123.0. If 7 digits were allowed, with 5 digits for a signed fractional part, we could represent the same number as `+.1234E3`, which is 123.4, and so forth.

Conceptually, binary representation of numbers is no different from decimal representation. The finite size imposes a limit on the range of integers and on the precision and range of floating point numbers. Binary representation is also tailored to facilitate the basic operations in hardware, such as addition and subtraction. For example, as we saw in Chapter 1, integers are typically represented in what is called the two's complement number system. However, one does not need to know the number system to realize that the limits on the range of values will be similar in nature and will depend on the sizes used to represent the numbers.

Recall that, in a computer, memory is organized as a sequence of bytes, each byte with an address, and storage is allocated in units of bytes. For example, if 1 byte is used for signed integers,

the range of values (in decimal) is -128 to 127; and unsigned integers have the range 0 to 255. If 2 bytes are used to represent signed integers, the range is -32768 to +32767; and 0 to 65535 for unsigned integers. If 4 bytes are used to represent integers, the range will be appropriately greater. Similarly for floating point numbers; with 4 bytes to represent floating point numbers, the precision is equivalent to about 7 significant decimal digits and a magnitude between approximately 10E38 and 10E-38. If more bytes are used for floating point numbers, the precision and the range are both appropriately greater.

So far we have used `char`, `int`, and `float` data types in our programs. Character data type is usually encoded as an ASCII integer value (signed or unsigned) in one byte of memory. Integers are at least two bytes in size, and floating point numbers are at least four bytes in size. C provides additional integer sizes and floating point data types that provide greater range and/or precision.

5.1.1   Signed and Unsigned Integer Types

For integer data types, there are three sizes: `int`, and two additional sizes called **long** and **short**, which are declared as `long int` and `short int`. The keywords `long` and `short` are called **subtype qualifiers**. The `long` is intended to provide a larger size of integer, and `short` is intended to provide a smaller size of integer. However, not all implementations provide distinct sizes for them. The requirement is that `short` and `int` must be at least 16 bits, `long` must be at least 32 bits, and that `short` is no longer than `int`, which is no longer than `long`. Typically, `short` is 16 bits, `long` is 32 bits, and `int` is either 16 or 32 bits.

Unless otherwise specified, all integer data types are *signed* data types, i.e. they have values which can be positive or negative. Recall, `char` types, without qualifiers, may be signed or unsigned depending on the implementation. However, all sizes of integers and `char` type may be explicitly qualified as `signed` or `unsigned`. (Unsigned numbers are always non-negative numbers).

For integers, `long`, `short`, and `unsigned` may be declared with the keyword `int` or without it. In C, whenever a data type is left out in a declaration, `int` is assumed by default. Here are some example declarations:

```
long int light_year;
short int n;
signed char ch;
unsigned char letter;
unsigned int age;
long distance;
short m, n;
unsigned memory_address;
unsigned long zip_code;
```

The data type of a constant, written directly into a program, is ascertained from the way it is written. Integer constants are written as a string of digits, optionally preceded by a unary positive

or a negative operator. Commas are not allowed. Decimal integer constants should be written without leading zeros, for example:

```
    29
  -173
     0
     1
  -525
 +7890
```

Alternate number systems may also be used to express integer constants in C programs. Octal numbers are written with a leading zero, and hexadecimal numbers are written with a preceding zero followed by the letter x or X:

```
Constants            Octal/Hexadecimal Integers

0234                 octal number 234
0101                 octal number 101
0x34                 hexadecimal number 34
0X1F                 hexadecimal number 1F
```

A constant to be represented as a long int may be explicitly written using the suffix l or L, as in:

```
123L
456781
```

Any integer constant that is too big to fit into the integer size is interpreted by the compiler as long.

Unsigned integers can be of all sizes, int, long, and short. The range of unsigned integers is 0 through 2^{k-1}, where k is the number of bits, so for 16 bits the maximum unsigned integer is 65535. Unsigned integer constants are written using the suffix, u or U:

```
0xFFFFU
123U
0777u
```

The two suffixes can be combined to write an unsigned long:

```
12345678UL
0X8FFF FFFFLU
```

5.1.2 Single and Double Precision Floating Point Numbers

Different sizes of floating point data can also be declared with the keywords `float` and `double`. The type specifier `double` is used to declare *double precision* floating point numbers. The size of `float` is typically 32 bits, and that of `double` is 64 bits. For greater precision, most scientific and engineering computation should be performed using the `double` data type. Furthermore, *extra precision* may be provided for floating point numbers by declaring them `long double`. (This may be the same as or more bits of precision as `double`, depending on implementation). Here are example declarations for floating point numbers:

```
float  x;
double GPR;
long double y;
```

Decimal `float` constants in programs have an integer part and a fractional part with a decimal point between them. They may also be written in scientific (or exponential) notation, i.e. a decimal number multiplied by a power of ten to indicate the actual position of the decimal point. Positive and negative numbers may be written with an explicit positive or negative unary operator.

```
123.789
0.5534
+9635.0000
-8942.3214
-0.765E5
1.4523e12
0.786345e-10
```

The last three numbers are written in exponential notation with the exponent of ten shown after the letter `e` or `E`. The exponent may be a positive or a negative integer. For clarity, always write `float` numbers with at least one digit before and one after the decimal point; for example, zero is `0.0` in `float` representation.

Floating point constants are taken to be of double precision type by default. Single precision floating point constants may be specified with a suffix `f` or `F`.

```
34.567f
3.141516F
```

Extra precision for constants may be written with the suffix `l` or `L`:

```
23456789.171819L
```

5.2   New Control Constructs

So far, we have seen all of the basic control constructs of the C language for calling functions, branching, and looping. In this section we introduce two new looping constructs that can be used in place of `while`; namely `for` loops and `do...while` loops.

5.2.1   The `for` Statement

The logic of the loops we have constructed so far has included three components: some form of initialization before the loop, a test for loop termination, and some form of data update within the body of the loop. We implemented these loops using three separate statements in the program, with a `while` statment forming the condition test and loop body. Another looping construct combines all three components of a loop in a single statement: the `for` statement.

The syntax for the `for` statement is:

    for (<expr1>; <expr2>; <expr3>) <statement>

The keyword, `for`, and the parentheses are required as shown. Notice the three expressions are separated by semi-colons (;). The semantics of the `for` statement is as follows. The expression, <expr1>, is evaluated once before the loop condition is tested for the first time; <expr2> is the loop condition which is evaluated prior to each execution of the loop body; and <expr3> is evaluated at the end of the loop body, just prior to testing the condition. The process repeats until the loop condition becomes False. The body of the loop is <statement>, which, as usual, may be any valid type of C statement; empty, simple, or compound. As with the `while` loop, if the loop condition evaluates to True, the loop body is executed; otherwise, if the loop condition evaluates to False, the loop is terminated, and control passes to the next statement following the `for` statement. In typical use, the expressions, <expr1> and <expr3> initialize and update a variable, respectively. Figure 5.1 shows the control flow for a `for` statement.

A `for` statement includes all the necessary features of a loop: an initialization expression, a loop condition, and an update expression. Thus, the following two forms of implementing a loop are equivalent:

```
        <expr1>;
        while (<expr2>) {
                <statement>     and         for (<expr1>; <expr2>; <expr3>) <statement>
                <expr3>;
        }
```

The `break` and `continue` statements can also be used in the body of a `for` statement, just as in a `while` statement. The use of a `for` statement or a `while` statement to implement a loop is a matter of choice, based on the logic of the algorithm. One advantage might be that writing a `for` statement reminds one that initialization and update expressions are usually necessary for a loop.
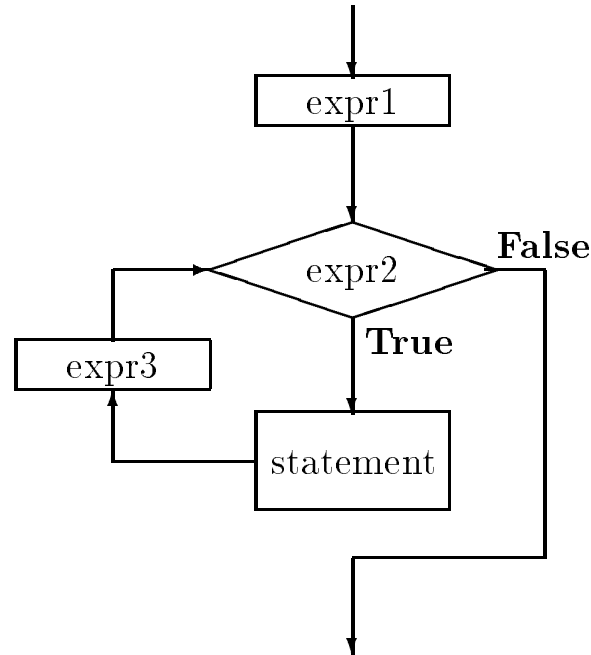
Figure 5.1: Control Flow of `for` Loop

An Example: Factorial

Let us consider an example task which may require a bigger range of integers than the one provided by `int` on many machines. The task is to determine a cumulative product from 1 to a positive integer, n. The product from 1 to n is called the factorial of n, written $n!$. The algorithm is very simple: read an integer `n`; call a function `fact(n)` which returns the factorial of `n`; print the result.

The function `fact()` merely needs to multiply a cumulative product variable, initialized to 1, by all integers from 1 through `n`:

```
initialize product to 1
repeat for values of i = 1, 2, 3,..., n
     product = product * i
return product
```

The variable, `product`, must be initialized to 1 before the loop, otherwise the cumulative product will be garbage. Each iteration brings us closer to the result. We will use a `for` statement to implement the iterative algorithm for a factorial function as shown in Figure 5.2. The `for` loop executes as follows. The first expression in parentheses is an initialization expression, i.e. `i` is initialized to 1. The second expression is the loop condition. If the second expression, `i <= n`, evaluates to True, then the loop body is executed. The third expression is the update expression; it is evaluated after the loop body is executed, and control then passes to the loop condition. In our example, the expression, `i = i + 1`, is evaluated to update the variable, `i`, after the loop body

```
/*    File: fact.c
      Program computes the factorial of integers using function
      fact().
*/
#include <stdio.h>
int fact(int n);
main()
{    int n;

     printf("***Factorial Program***\n");
     printf("Type positive integers, EOF to terminate\n");
     while (scanf("%d", &n) != EOF)
          if (n <= 0)
               printf("%d typed, type a positive integer\n", n);
          else
               printf("Factorial of %d is %d\n", n, fact(n));
}


/*    Function computes factorial of n using a for loop. */
int fact(int n)
{    int i, product;

     product = 1;
     for (i = 1; i <= n; i = i + 1)
          product = product * i;
     return product;
}
```

Figure 5.2: Code for factorial

is executed. The loop condition is then tested, and the process repeats until the loop condition becomes False. The above loop executes for i = 1, 2, 3, ..., and n and the variable product accumulates the factorial value of $1 * 2 * 3 * ... * n$.

The driver uses a while condition:

```
     (scanf("%d", &n) != EOF)
```

where scanf() reads an integer item if possible and stores it in n. The value returned by scanf() is then compared with EOF and if the value returned is NOT EOF, the loop executes. As soon as scanf() returns EOF, the loop is terminated. The while expression serves both to read an item and to check if the returned value is EOF. The loop body tests the value of n; if it not a positive integer, the user is asked to retype a positive number; otherwise, the value of fact(n) is printed.

Here is a sample session run on an IBM PC:

```
***Factorial Program***
Type positive integers, EOF to terminate
4
Factorial of 4 is 24
5
Factorial of 5 is 120
-3
Negative number -3 typed, type positive integers
6
Factorial of 6 is 720
7
Factorial of 7 is 5040
8
Factorial of 8 is -25216
^Z
```

The cumulative product in the factorial function grows very fast with n. For moderately large values of n, the cumulative product *overflows* the int type object; the number is too large for the size of the object. When this occurs, the results are meaningless. Usually, an overflow is indicated when a program, working correctly for smaller numbers, gives ridiculous results for larger numbers. In the case of the factorial function, the first sign of trouble is a negative result for the factorial of 8. We know the result must be positive since we are multiplying only positive numbers. What has happened is the result has overflowed into the sign bit resulting in a negative integer. If factorial of larger numbers is desired, a long int variable should be used for the variable product as well as for the function fact. Here is a revised version of the factorial function.

```
/*   Function computes a long factorial of n using a for loop. */
long longfact(int n)
{    long int product;
     int i;

     product = 1;
     for (i = 1; i <= n; i = i + 1)
          product = product * i;
     return product;
}
```

We must keep several things in mind when using the function, longfact(), in the driver program. In the calling function, if the value returned by longfact() is saved, it must be assigned to a long integer; otherwise, a long result would be converted to int by dropping higher order bits and the result would be meaningless. In addition, to print the long value of longfact(), the conversion specifier must be qualified by the prefix l:

```
printf("Factorial of %d is %ld\n", n, longfact(n));
```
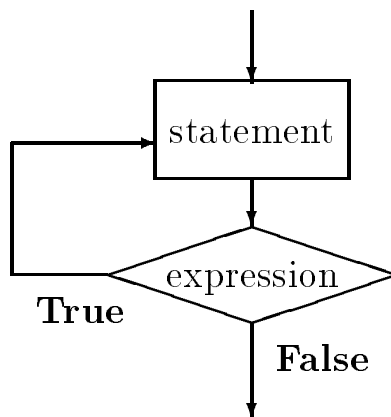
Figure 5.3: Control Flow of `do...while` Loop

The conversion specifier, `%ld`, prints a long decimal integer.

This example has shown a case where the size of the type `int` is smaller than the type `long`, as it is in some implementations. The situation could be corrected by using a larger size data type to accumulate the factorial. However, even this type has limitations; the factorial of 13 will overflow the size of a long integer. The only possibility provided for even larger numbers is to use a floating point representation, which has a larger range, at the expense of loss of precision.

5.2.2   The `do...while` Statement

In `while` statements and `for` statements, the condition is tested for each iteration *before* the loop body is executed. Thus, it is possible that the loop may not be executed even once if the loop condition evaluates to False the first time. The C language provides another looping construct which guarantees that the body will be executed at least once: the `do...while` statement. The loop condition is tested *after* the body is executed, and the loop continues or terminates depending on the condition value. The syntax for the `do...while` statement is:

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

Figure 5.3 shows the control flow for this construct. As with the other loop constructs, the `break` and `continue` statements can also be used with the `do...while` statement. The choice of a loop construct depends on the program logic. There are situations when one construct may be preferable to another.

An Example: Square Root

Programs are often written to find a solution (or solutions) to an algebraic equation; for example:

$$y^2 - x = 0$$

Here, the solution for the variable, y, is the square root of x. In general, such solutions are real numbers, and as we have seen, floating point representations of real numbers use a finite number of bits, and are therefore limited in the precision of the result. Solutions to most numeric problems can never be exact (all solutions are precise only up to a certain number of decimal digits) but the result may be sufficiently close to the real solution to be acceptable.

One important numeric computation method to find solutions to equations involves successive approximations. This method starts with a *guess* for the solution to the problem, and tests if the guess satisfies the equation. If the guess is *close enough* for a solution, it is accepted and computation terminates; otherwise, the guess is *improved*, i.e. brought closer to the solution and the process is repeated. After each iteration, the guess is closer and closer to the solution, until it is acceptably close enough.

One successive approximation algorithm we will use is Newton's method to compute the square root of a number, x. Newton's method starts with an arbitrary guess, and if it is not good enough, it is improved by averaging the guess with $x/guess$. The process continues until the guess is close enough. Here is an example of the process for square root of 9.0:

guess	$x/guess$	Average
1.0	9.0	$(1.0 + 9.0)/2.0$
5.0	1.8	3.4
3.4	2.647	3.023
3.023	. . .	

In just three iterations, we have arrived close to the square root of 9.0 (which is 3.0). We will say a guess is close enough to the solution, if x and the square of *guess* differ by a small value, say 0.001, or less. The algorithm is simple:

```
begin with an initial guess
repeatedly do the following
     improve the guess
while it is not close enough
```

We will start with an arbitrary guess, say 1.0, for the square root of the number, x. In a loop, each iteration improves the guess of the square root of x until the guess is close enough. In our implementation, we assume two functions: one to test if a guess is close enough, and the second to improve the guess. This algorithm works for any successive approximation method; the only difference would be how to improve the guess, and how to check the guess for closeness to the solution. Here is the code fragment for square root using a `do...while` statement:

```
guess = 1.0;
do
     guess = improve(guess, x);
while (!close(guess, x))
```

The body of the loop follows the keyword `do`. The loop body is executed and then the `while` expression is tested for True or False. If it is True, the loop is repeated; otherwise, the loop is terminated. The above loop body calls on a function `improve()` to improve the guess and the condition is then tested to see if the improved guess is close enough by the function `close()`.

As we said, the difference between `do...while` and the other loop constructs is that in this case the loop is executed at least once; `while` loops and `for` loops may be executed zero times if the loop condition is initially False. In the case of successive approximations, we always expect the initial guess to need improvement; so, the loop must be executed at least once.

Figure 5.4 shows the implementation of the driver. The source file includes a header file `mathutil.h` that declares the function prototypes for `close()`, `improve()`, and other functions defined in a source file, `mathutil.c`, shown in Figure 5.5. The two source files `sqroot.c` and `mathutil.c` must be compiled and linked to create an executable file. Here is `mathutil.h`:

```
/* File: mathutil.h */
/* File contains prototypes for functions defined in mathutil.c */
double improve(double guess, double x);
int close(double guess, double x);
double absolute(double x);
```

Notice we have used the type, `double` for the parameters and return values of the functions because precision is important in successive approximation algorithms. It is best to use double precision in all such computations. We have also included the header file, `tfdef.h`, which defines the symbolic constants `TRUE` and `FALSE`.

The program driver uses a loop to read a positive, double precision number into `x` using the conversion specification `%lf`. (When a double precision number is printed, conversion specification is still `%f` since a printed double precision floating point number looks the same as a single precision number). If the number read into `x` is negative or zero, a message is printed and the loop is repeated until a positive number is read. We have used the `do...while` construct here, since we know that the loop must be executed at least once to get the desired data.

Next, `guess` is initialized to 1.0 and the loop body improves `guess` We have included a debug statement to print the value of the improved guess during program testing. The loop repeats until `guess` is close enough to be an acceptable solution.

We still need to write the functions `improve()` and `close()`. The function `close()` tests if the absolute value of the difference between the square of `guess` and `x` is small enough. We will use a function, `absolute()`, that returns the absolute value of its argument. Figure 5.5 shows `close()` and `absolute()` in the source file, `mathutil.c`. Some of the functions defined in this source file

```
/*    File: sqroot.c
      Other Files: mathutil.c
      Header Files: tfdef.h, mathutil.h
      Program computes and prints square roots of numbers. Uses Newton's
      method to compute square root of x: Start with any guess. Test if
      it is acceptable. If not, improve guess by averaging it with x/guess.
*/
#include <stdio.h>
#include "tfdef.h"
#include "mathutil.h"
#define DEBUG
main()
{     int i;
      double x, guess;

      printf("***Square Root Program: Newton's Method***\n\n");
      printf("Type a positive number: ");
      do {
          scanf("%lf", &x);
          if (x <= 0)
              printf("%f typed, type a positive number\n", x);
      } while (x <= 0);
      guess = 1.0;
      do {
          guess = improve(guess, x);         /* improve guess.    */
          #ifdef DEBUG                                /* debug stmt    */
              printf("guess = %f\n", guess);    /* Print guess.  */
          #endif                                    /* end of debug  */
      } while (!close(guess, x));         /* terminate if guess is close */
      /* exit loop if guess is close enough */
      printf("Sq.Rt. of %f is %f\n", x, guess);    /* Print sq. rt. */
}
```

Figure 5.4: Code for Square Root

```
/* File: mathutil.c */
#include <stdio.h>
#include "tfdef.h"
#include "mathutil.h"
/* Tests if square of guess approximately equals x. */
int close(double guess, double x)
{
    if (absolute(guess * guess - x) < 0.001)
        return TRUE;
    else
        return FALSE;
}

/* Returns absolute value of x. */
double absolute(double x)
{
    if (x < 0)
        return -x;
    else
        return x;
}

/* Returns average of guess and x / guess. */
double improve(double guess, double x)
{
    return (guess + x / guess) / 2;
}
```

Figure 5.5: Code for Math Utilities

are also called within it, e.g. `absolute()`, so we have included `mathutil.h` in this source file, as well as `tfdef.h`, which defines `TRUE` and `FALSE`. Finally, we write the function `improve()` which merely returns the average of `guess` and `x / guess`.

Sample Session:

```
***Square Root Program***

Type a number:  16
guess = 8.500000
guess = 5.191176
guess = 4.136665
guess = 4.002257
guess = 4.000000
Sq.Rt.  of 16.000000 is 4.000000
```

The debug statement shows how guess is changed at each step. Once we are satisfied with the program, we can remove the definition of `DEBUG`.

Next, we modify our program to encapsulate it into a function, `sqroot()`, and to provide user control over the precision desired for the solution instead of building it into the function, `close()`. The `sqroot()` function requires two arguments, a number and an acceptable error in the solution. We also require a new function `close2()` that checks if a given guess is close enough to a solution with a specified margin of error. With this modification, it is not necessary to use `double` for numbers in `main()`. Only the actual computations need to be `double` type for greater precision. Figure 5.6 shows the revised driver in which `float` numbers are used in `main()` and the function `sqroot()` is called to find the square root. Figure 5.7 shows the prototypes added to `mathutil.h` and the new functions in `mathutil.c`. The driver simply repeats the following loop: read a number; if the number is negative, continue the loop; otherwise, call `sqroot()` to find the square root of the number within specified margin; print the value. The function `sqroot()` merely starts with a guess and improves it in a loop until it is within an allowable margin of error. The final acceptable guess is returned. The function `close2()` tests if a guess is close to the solution within a specified error.

In `main()`, numbers are read into `float` variables, so when arguments are passed to `sqroot()`, they are cast to `double`. Likewise, the returned `double` value is cast to `float` before assigning it to the variable `root`. Here is the statement that uses cast operators to convert types:

```
root = (float) sqroot((double) x, 0.001);
```

Recall that a floating point constant is always assumed to be of type `double`. If function prototypes are declared, we don't have to convert the types explicitly by cast operators, the compiler will take care of that for both the arguments and the returned value. However, the explicit cast operators improve readability by showing that conversions are taking place.

Sample Session:

```
/*   File: sqrt2.c
     Other Files: mathutil.c
     Header Files: tfdef.h, mathutil.h
     Program computes and prints square roots of numbers until the end of
     file. Uses Newton's method to compute the square root of x to within a
     specified error margin.
*/
#include <stdio.h>
#include "tfdef.h"
#include "mathutil.h"
main()
{    int i;
     float x, root;

     printf("***Square Root Program***\n\n");
     printf("Type positive numbers, EOF to quit: ");
     while (scanf("%f", &x) != EOF) {
          if (x <= 0) {
               printf("%f typed, type positive numbers \n");
               continue;
          }
          root = (float) sqroot((double) x, 0.001);
          printf("Sq.Rt. of %f is %f\n", x, root);
     }
}
```

Figure 5.6: Modified Square Root Driver

```
/* File: mathutil.h - continued */
double sqroot(double y, double error);
int close2(double g, double y, double error);



/* File: mathutil.c - continued */
/* Uses Newton's method to compute square root within the margin
   allowed by error.
*/
double sqroot(double y, double error)
{    double guess = 1.0;

     do
          guess = improve(guess, y);    /* improve guess. */
     while (!close2(guess, y, error));  /* while guess not close */
     return guess;                      /* when close enough, return guess.*/
}

/* Tests if square of g equals y within the error limits specified. */
int close2(double g, double y, double error)
{
     if (absolute(g * g - y) < error)
          return TRUE;
     else
          return FALSE;
}
```

Figure 5.7: Modified Square Root Utilities

```
***Square Root Program***

Type positive numbers, EOF to quit:   16
Sq.Rt.  of 16.000000 is 4.000000
13
Sq.Rt.  of 13.000000 is 3.605551
19
Sq.Rt.  of 19.000000 is 4.358902
25
Sq.Rt.  of 25.000000 is 5.000023
^D
```

The last example shows the square root of 25.0 to be slightly different from the correct value of 5.0, but within our allowed error of 0.001. It must be remembered that floating point representation cannot be exact due to the finite number of bits used. Therefore, if the error specified were very small, it may not be possible to arrive at an answer with the desired accuracy. That is, the guess may never converge to a value such that `close2()` returns True and the loop in `sqroot()` would never terminate. In successive approximations algorithms, one must guard against possible lack of convergence such as by putting a limit on the number of loop iterations allowed.

In Chapter 6 we will see that standard library functions are available to compute the square root and the absolute value of a number. Our emphasis here has been to illustrate program development using just the basics of a programming language, viz. expressions including assignments, branching, and looping.

5.3   Scalar Data Types

All of the data types we have seen so far, `char`, `int`, `short long`, `float`, and `double` are called **scalar** (or base) data types because they hold a single data item. (Chapters 7 and 12 describe compound data types provided in C). There are two other scalar types in the language: `enum` and `void` which are described in this section. We will refer to `float` and `double` types as floating point types and to all sizes of integers, `char` and `enum` types as integral types. In addition, we describe how a user defined type may be declared.

5.3.1   Data Type `void`

The data type `void` actually refers to an object that does not have a value of any type. The most common example of its use is when we define a function that returns no value. For example, a function may only print a message and no return value is needed. Such a function is used for its side effect and not for its value. In the function declaration and definition, it is necessary to indicate that the function does not return a value by using the data type `void` to indicate an empty type, i.e. no value. Similarly, when a function has no formal parameters, the keyword `void`

is used in the function prototype and header to signify that there is no information passed to the function.

Here is a simple program using a message printing function which takes a **void** parameter and returns type **void**:

```
/*   File: msg.c
     This program introduces data type void.
*/

void printmsg(void);

main()
{

    /* print a message */
    printmsg();
}

/* Function prints a message. */
void printmsg(void)
{
    printf("****HOME IS WHERE THE HEART IS****\n");
}
```

No parameters are required for the function, `printmsg()`, and it returns no value; it merely prints its message. In the function call in `main()`, parentheses must be used without any arguments. Observe that no return statement is present in `printmsg()`. When a function is called, the body is executed and, when the end of the body is reached, program control returns to the calling function. Such a return from a called function without a **return** statement is often called returning by *falling off the end*. There are times when it is necessary to return from a **void** function before the end of the body. In such case, a **return** statement, with an empty expression may be used to return nothing:

```
void printmsg(void)
{
    printf("****HOME IS WHERE THE HEART IS****\n");
    return;
}
```

A **return** statement can also be used elsewhere in the body to return control immediately to the calling function. Consider a function which prints the values of its arguments if they are all positive; otherwise it does nothing:

```
    void func(int x, in y)
```

```
    {
        if (x <= 0 || y <= 0)
            return;
        printf("x = %d, y = %d\n", x, y);
    }
```

If either of the arguments is not positive, the function returns to the calling function. If it does not return, then it prints the values of the arguments.

The use of `void` for a function returning no value is not strictly necessary. We could declare the function as being type `int` (or any other type) and simply not return any value and never use the value of the function in an expression. However, the `void` declaration makes the nature of the function explicit to someone reading the code and may allow the compiler to generate more efficient object code.

5.3.2   Enumeration

The data type, `enum` (for enumeration) also allows improvement in program clarity by specifying a list of names, the enumeration constants, which are associated with constant integer values. It is similar to using `#define` directives to define constant integer values for a set of symbolic names; however, with `enum` the compiler can generate the values for you, and may check for proper use of `enum` type variables. A variable of `enum` type is declared as follows:

```
    enum { FALSE, TRUE } flag;
```

The variable, `flag`, is defined here to be of a type which can take on the two enumerated constant values, `FALSE` and `TRUE`. Normally, enumeration constants are identifiers whose values start at zero and increase in sequence: here, `FALSE` is 0, and `TRUE` is 1. However, the enumeration can have explicit constant values specified in the enumeration:

```
    enum { SUN = 1, MON, TUE, WED, THU, FRI, SAT } day;
```

Here, `SUN` is associated with value 1, and the rest of the names have values in increasing sequence: `MON` is 2, `TUE` is 3, and so on until `SAT` which is 7. The variable, `day` can hold any of the enumerated values.

An enumeration type can be given a *tag*, i.e a name which can be used later to declare variables of that tagged enumeration type. For example, we can name an enumeration:

```
    enum boolean { FALSE, TRUE };
```

where the name `boolean` can then be used to declare variables of that enumeration type:

```
enum boolean flag1, flag2;
```

This declaration defines variables, `flag1` and `flag2`, which are of a type specified by the boolean enumeration; that is, `flag1` and `flag2` can have values `FALSE` or `TRUE`. It is also possible to specify a tag and declare variables in the same declaration:

```
enum boolean {FALSE, TRUE} done;
enum boolean found;
```

The first declaration specifies a tag, `boolean`, for the enumeration as well as declaring a variable, `done` of this type. The second declaration defines a variable, `found`, of the enumeration `boolean` type. Here is a function, `digitp()`, that returns a `boolean` value to the calling function. (The calling function must also declare the enumeration in order to use the returned value correctly).

```
enum boolean { FALSE, TRUE };

enum boolean digitp(char c)
{
    if (c >= '0' && c <= '9')
        return TRUE;
    else
        return FALSE;
}
```

Remember, the *value* of an `enum` type variable is an integer. An enumerated data type is primarily a convenience for writing the source code; information about the symbolic names are not retained at run time. For example, if we were to execute a statement:

```
printf("digitp returns %d\n",digitp('0'));
```

it would print

```
digitp returns 1
```

NOT

```
digitp returns TRUE
```

However, some symbolic debuggers may use the enumerated names for displaying debugging information.

5.3.3   Defining User Types: `typedef`

The C language provides a facility for defining *synonyms* for data types to make programs more readable. New data types that are equivalent to existing data types may be created using the `typedef` declaration. The syntax is:

> typedef <existing-type-specifier> <new-type-specifier>;

The scope of a type definition is from the point of definition to the end of the source file. Variables can then be defined in terms of these new types. For example, variables used to represent values of age of people or objects can be defined to be of a new type, `age`.

```
typedef int age;
age yrs;
```

The variable `yrs` can have `age` type values. In this case, the primary difference is that we can have more meaningful names for data types than the generic name `int`.

A `typedef` definition is also commonly used to "hide the details" of more complicated declarations:

```
typedef enum { FALSE, TRUE } boolean;

boolean flag;
```

The type definition defines data type, `boolean` which is a synonym for an enumerated type consisting of two constant values `FALSE` and `TRUE`. Variables of type `boolean` can now be defined, and they can be assigned one of the enumerated values. In fact, the name, `boolean`, can be used like any other data type. Functions can have `boolean` parameters and can return `boolean` values. For example, we could write:

```
flag = TRUE;

if (flag)
    printf("Flag is true\n");
```

Let us consider the task of a simple calculator. It should read two numbers and then read an operator that is to be applied to the operands. The operator should be applied to the operands and the result printed. (When an operator appears after the operands, the expression is said to be in **postfix form**). The algorithm for a postfix calculator is:

```
repeat until end of file or error in reading numbers
     read two numbers and an operator
     apply operator to the numbers and get result
     print result
```

The program must make sure that two valid numbers and an operator are read correctly. We will ensure that two numbers are read correctly by examining the value returned by `scanf()`. The buffer will then be scanned and flushed until a valid operator is found. The program is shown in Figure 5.8.

The `while` loop continues until `scanf()` is unable to read two numbers. If `scanf()` reads two numbers, it returns a value of 2, and the loop is executed. In the loop, we use `get_operator()` to get a valid operator. The function, `get_operator()` will scan each new character in the keyboard buffer until an acceptable operator is found. Once an operator is read, an error flag of type `boolean` is initialized to `FALSE`.

A switch statement is used to determine the result of applying the operator to the operands. The division operator can lead to trouble if `oprnd2` is zero; divide by zero is a fatal error and the program would be aborted. We trap this error by testing for a zero value of `oprnd2`, in which case we set `error` to `TRUE`. If there is no error, the result is printed ; otherwise, an error message is printed. The loop repeats until `scanf()` does not read 2 `float`s (including detecting `EOF`).

The function `get_operator()` consists of a loop that continues to read a character until a valid operator is read; skipping over any white space and any erroneous characters. It uses a `boolean` type function, `operatorp()`, to test if an argument is an acceptable operator. Figure 5.9 shows the required functions.

Sample Session:

```
***Postfix Calculator***

Type two numbers, followed by an operator:  +, -, *, or /
EOF to quit
12      12
    +
12.000000 + 12.000000 = 24.000000
50      0
/
Runtime error:  50 / 0
^D
```

We have purposely used a lot of white space to show that the calculator functions correctly.

5.4   Operators and Expression Evaluation

Once we can declare data to be the type and size with the appropriate precision for our task, we would like to perform operations with the data. We have already discussed some of the basic C operators, and in this section we provide the complete precedence table for all C operators. We

```
/*    File: calc.c
      This program is a postfix calculator. Two operands followed by an
      operator must be entered. The program prints the result. The program
      repeats until end of file.
*/
#include <stdio.h>
typedef enum { FALSE, TRUE } boolean;
char get_operator(void);
boolean operatorp(char c);

main()
{     float oprnd1, oprnd2, result;
      char c;
      boolean error;

      printf("***Postfix Calculator***\n\n");
      printf("Type two numbers, followed by an operator: +, -, *, or /\n");
      printf("EOF to quit\n");

      while (scanf("%f %f", &oprnd1, &oprnd2) == 2) {
            c = get_operator();
            error = FALSE;

            switch(c) {
                  case '+': result = oprnd1 + oprnd2; break;
                  case '-': result = oprnd1 - oprnd2; break;
                  case '*': result = oprnd1 * oprnd2; break;
                  case '/': if (oprnd2)
                                    result = oprnd1 / oprnd2;
                              else
                                    error = TRUE;
                              break;
            }

            if (error == FALSE)
                  printf("%f %c %f = %f\n", oprnd1, c, oprnd2, result);
            else
                  printf("Runtime error: %f %c %f\n", oprnd1, c, oprnd2);
      } /* end of while loop */
} /* end of program */
```

Figure 5.8: Code for Simple Postfix Calculator

```
/* File: calc.c - continued */
/* Gets one of the allowed operator, +, -  , *, /. */
char get_operator(void)
{    char c;

     while ((c = getchar()) && operatorp(c) != TRUE)
         ;
     return c;
}


/* Function tests if c is one of the operators +, -, *, /. */
boolean operatorp(char c)
{
     switch(c) {
         case '+':
         case '-':
         case '*':
         case '/': return TRUE;
         default: return FALSE;
     }
}
```

Figure 5.9: Code for get_operator()

present a few new operators here, and others shown in the table will be discussed in detail in later chapters.

5.4.1 Precedence and Associativity

The data type and the value of an expression depends on the data types of the operands and the order of evaluation of operators which is determined by the precedence and associativity of operators. Let us first consider the order of evaluation. When expressions contain more than one operator, the order in which the operators are evaluated depends on their precedence levels. A higher precedence operator is evaluated before a lower precedence operator. If the precedence levels of operators are the same, then the order of evaluation depends on their associativity (or, grouping). In Chapter 2 we briefly discussed the precedence and associativity of arithmetic operators. Table 5.1 shows the precedence levels and associativity of all C operators.

In the table, there are 15 precedence levels 0 through 14: higher level implies higher precedence. The precedence levels of operators are separated by solid lines with operators within solid lines having the same precedence level. For example, binary arithmetic operators *, /, and % have the same precedence level which is higher than binary +, and −. Observe that the precedence of the assignment operator is lower than all but the "comma" operator (described below). This is in accordance with the rule that the expression on the right side of an assignment is evaluated first, and then its value is assigned to the left hand side object. On the other hand, "function call" has the highest precedence, since a function *value* is treated like a variable reference in an expression. In any expression, parentheses may be used to over ride the precedence of the operators — innermost parentheses are always evaluated first. The precedence of binary logical operators is lower than that of binary relational operators; that of binary relational operators is lower than that of binary arithmetic operators, and so forth. The unary NOT operator has a precedence higher than that of all binary operators.

When operators of the same precedence level appear in an expression, the order of evaluation is determined by the associativity. Except for the assignment operator, associativity of most binary operators is left to right; associativity of the assignment operator and most unary operators is right to left. Consider the following program fragment:

```
int x = 10, y = 7, z = 20;
```

`x = -20 + 10 * 5;`	By the precedence, the unary minus $(-)$ is evaluated first; followed by the multiplication $(*)$ and then the addition. So the expression is evaluated as `(-20) + (10 * 5)` and finally the result is assigned to `x` which now has the value 30.
`x = x / y * z;`	Here the / and * have the same precedence, so by associativity are evaluated left to right: $(x/y)*z$. This is $30/7*20$, or $4*20$ (integer division); so 80 is assigned to `x`.

	Operator	Associativity	Precedence
()	Function call	Left-to-Right	Highest 14
[]	Array subscript		
.	Dot (Member of structure)		
$->$	Arrow (Member of structure)		
!	Logical NOT	Right-to-Left	13
~	One's-complement		
$-$	Unary minus (Negation)		
++	Increment		
$--$	Decrement		
&	Address-of		
*	Indirection		
(type)	Cast		
sizeof	Sizeof		
*	Multiplication	Left-to-Right	12
/	Division		
%	Modulus (Remainder)		
+	Addition	Left-to-Right	11
$-$	Subtraction		
<<	Left-shift	Left-to-Right	10
>>	Right-shift		
<	Less than	Left-to-Right	8
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equal to	Left-to-Right	8
!=	Not equal to		
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
|	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
||	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, += *=, etc.	Assignment operators	Right-to-Left	1
,	Comma	Left-to-Right	Lowest 0

Table 5.1: Precedence and Associativity Table

`x / y / z`	Again, associativity causes the operators to be evaluated left to right. (80/7)/20, i.e. 11/20 or 0. (No assignment is made here).
`x % y % z`	Evaluated as (80%7)%20; 3%20 or 3.
`x = y = 10;`	The assignment operator associates right to left; so `y` is assigned 10, and then the result value, 10, is assigned to `x`.
`x + y <= 2 * z`	The highest precedence operator is $*$, so it is evaluated first, followed by $+$, and finally the comparison operator, $<=$. The result, 87, is not less than or equal to 40, so this expression evaluates to False, namely 0.
`(x + y >= 2 * z) && (x - y != z)`	The parentheses force the logical operator `&&` to be evaluated last. Its left operand is similar to the last expression; only the result is now True, or 1. The right operand evaluates the subtraction followed by the comparison, not equal. Since 73 is not equal to 20, the result is True, and therefore, the entire expression is True, or 1.

When a logical operator is used in an expression, the entire expression is not evaluated if the result of the entire logical expression is clear. For example,

```
(x > 0) && (y > 0)
(x > 0) || (y > 0)
```

In the first expression, if `x > 0` is False, there is no need to evaluate the second part of the logical AND expression since the AND operation will be False. Similarly, in the second expression, the logical OR expression is True if the first part, `x > 0`, is True; there is no need to evaluate the second part. C evaluates only those parts of a logical expression that are required in order to arrive at the result of the expression.

When in doubt as to the order of evaluation within an expression, parentheses may be used to ensure evaluation is performed as intended.

5.4.2   The Data Type of the Result

The data type of an expression value depends on the operators and the types of operands. If the operands are all of the same type, the result is of that same type. When there are operands of mixed type in an assignment expression, the right hand side is always converted to the data type of the object on the left hand side. This follows common sense since the type of the object on the left of an assignment is fixed and cannot be changed. When any other binary operator is applied

to operands of mixed type, the operand of a type with lower range is converted to the type of the higher range operand before the operator is applied; and the result is of the higher range type. Of course, values of characters in an expression are considered to be `int` type. Again, some examples will illustrate:

```
int n = 3, m = 2;
long large;
float x = 9.0, y = 5.0;
double z = 4.0;
```

`large = n;`	The *integer value* of `n` is converted to `long` and assigned to `large`.
`large = n / m;`	Since `n` and `m` are both type `int`, integer division is performed (3 / 2 which is 1); and then converted to `long` (1L) which is assigned to `large`.
`large = n / x;`	Since `x` is a `float`, the *integer value* of `n` is converted to `float` and real division (3.0 / 9.0) is performed yielding 0.33. This result is then converted to a long integer (by truncating), namely 0L, and assigned to `large`.
`z = z * y;`	Because `z` is type `double`, the *value* of `y` is converted to `double` and the double precision result of `z * y` is assigned to `z`.
`n / x + z / y;`	In the first division, `n / x`, since `x` is type `float` the division will be done at `float` precision by first converting the value of `n`, yielding a `float` result. The second division will be performed using double precision because `z` is a `double`, by first converting `y` to a `double`. The addition is now of a `float` and a `double`, so the left operand is first converted to `double` yielding a `double` result. This is equivalent to:

```
(double) ((float) n / x) + z / (double) y;
```

As with the precedence and associativity rules, when in doubt as to the type and/or precision of an expression evaluation, cast operators may be used to force conversions to the desired type. Remember, only *values* of variables are converted for the purpose of computation, NOT the variables themselves.

5.4.3   Some New Operators

In Table 5.1 there are several operators which we have not yet discussed. Some of these are described below; the remainder will be delayed until later chapters when we discuss the appropriate data types.

Increment and Decrements Operators

A common operation in many programs is to increase or decrease a variable value by one; for example, this is how we keep a count of how many times a loop is executed. C provides a "shorthand" way of performing this operation with special increment and decrement operators, `++` and `--` respectively. These are unary operators requiring one operand and may be used either as **prefix** or **postfix** operators meaning they either precede or follow their operands. In postfix form, `x++` increases the value of `x` by one, and `y--` decreases the value of `y` by one. Likewise, in prefix form, `++x` increases the value of `x` by one, and `--y` decreases the value of `y` by one. However, there is a difference between the prefix and postfix operators. In the case of prefix operators, the operation is performed first *and then* the expression evaluates to the new value of its operand. For postfix operators, the expression first evaluates to the `current` value of the operand and then the operators are applied. For example, if `x` is 1, the expression `++x` first increments `x` to 2 and then evaluates to the value 2. On the other hand, again if `x` is 1, the expression `x++` first evaluates to the value of `x`, namely 1, and then increments `x` to 2. Here is a code fragment showing the use of the increment and decrement operators:

```
int x, y, z1, z2, z3;

x = 4;
y = 4;
```

`x++;`	The value of `x` is incremented to 5. The value of this expression is 4, but is discarded.
`y--;`	The value of `y` is decremented to 3. The value of this expression is also 4, but is also discarded.
`z1 = x++ - y++;`	The expression `x++` evaluates to the current value of `x`, 5, and then `x` gets the value 6. Likewise, `y++` evaluates to 3 and then `y` is incremented to 4. The variable `z1` gets the value of `5 - 3`, or 2.
`z2 = ++x - ++y;`	First, `x` is incremented to 7, and `++x` evaluates to 7. Likewise `++y` increments to `y` to 5 and evaluates to 5, so `z2` gets the value of `7 - 5` or 2.

```
    z3 = x++ + --y;
```
The expression, `x++`, evaluates to 7 and then increments `x` to 8. The expression, `--y`, decrements `y` to 4 and evaluates to 4. So `z3` gets the value of `7 + 4`, or 11.

The value of

```
    ++x - x++
```

is implementation dependent. A compiler may either evaluate the first term first or the second term first. It is therefore not possible to say what the expression will evaluate to. For example, assume that x is initially 1. If the first expression is evaluated first, then the expression is:

```
    2 - 2
```

i.e. 0, and `x` is 3. On the other hand, if the second term is evaluated first, then the expression is:

```
    3 - 1
```

i.e. 2, and `x` is 3.

Increment and decrement operations can just as well be written as assignment expressions:

```
    x = x + 1;
    y = y - 1;
```

The use of increment and decrement operators does not accomplish anything that cannot be done by appropriately placed assignments. These operators were designed to be used with machines that have increment and decrement registers; in which case the compiler can take advantage of these registers and improve the performance of the program. However, many machines today do not have these registers, so most compilers translate expressions with increment and decrement operators in exactly the same manner as they do assignment expressions, but these operators remain as a "shorthand" syntax for compact programs.

The syntax of the increment and decrement operators is:

```
    ++ <Lvalue>
    -- <Lvalue>
    <Lvalue> ++
    <Lvalue> --
```

The operand must be and <Lvalue>, i.e. a location into which a value can be placed. (So far, we have seen that only a variable name may be used as an <Lvalue>. We will see other possibilities

Composite   Equivalent

```
x += 5;      x = x + 5;
y -= 12;     y = y - 12;
x *= 3;      x = x * 3;
y /= 5;      y = y / 5;
x %= 7;      x = x % 7;
```

Table 5.2: Composite Assignment Operators and Their Equivalents

in Chapter 6). The precedence and associativity of increment and decrement operators is given in Table 5.1. Here are some examples of their use in program code:

```
for (i = 0; i < MAX; i++)          The message, This is a test will be printed MAX
    printf("This is a test\n");    times.

n = 0;                             The expression n++ evaluates to the value of n
while (n++ < 10)                   before it is incremented. The loop will print the
    printf("Value of n is %d\n", n);values 1,2, ...,10 for n.

n = 0;                             The expression ++n evaluates to the value of n
while (++n < 10)                   after it is incremented. The loop will print the
    printf("value of n is %d\n", n); values 1,2, ...,9 for n.
```

Composite Assignment Statements

The above operators provide a "shorthand" way of increasing or decreasing a variable by one; but sometimes we would like to increase or decrease (or multiply, divide or mod) by some other value. C provides "short hand" operators for these as well, called the **composite assignment operators**. These operators and their equivalent are shown in Table 5.2.

The general syntax of a composite assignment operator is:

<Lvalue> <op>= <expression>

where <op> may be one of the binary arithmetic operators, +, -, *, /, or %. The left operand of these operators must be an <Lvalue>, but the right operand may be an arbitrary <expression>.

Again, there is no particular advantage in using the composite assignment operators over the simple assignment operator except that they produce a somewhat more compact program. The precedence and grouping for composite assignment operators given in Table 5.1 shows they are the same as the assignment operator. Figure 5.10 shows the factorial function (see Figure 5.2) using these new operators.

```
/*    File: mathutil.c - continued */
/*    Function returns long factorial of n. */
long factcomp(int n)
{     int i;
      long prod;

      prod = 1;                     /* initialize */
      for (i = 1; i <= n; i++)      /* loop from 1 to n */
          prod *= i;                /* compute cumulative product */
      return prod;                  /* return product */
}
```

Figure 5.10: Factorial Function Using Composite Operators

Conditional Expression

Sometimes in a program we would like to determine the value of an expression based on some condition. For example, if we had two variables, x and y, and we wanted to assign the larger value to the variable, z. We could write and if statement to perform this task as follows:

```
if (x < y)  z = y;
else z = x;
```

Another way of stating this in words is that z should be assigned the value of y if x < y or x, otherwise. The (operator) symbols ? and : may be used to form such a conditional expression as follows:

```
z = x < y ? y : x;
```

The expression to the right of the assignment operator is evaluated first as follows. If x < y, the expression evaluates to the value of the expression after ?, i.e. y. Otherwise, it evaluates to the value of the expression after :, i.e. x. In other words, the expression evaluates to the larger of x and y which is then assigned to the variable, z.

As another example, we can write an expression that evaluates to the absolute value of x:

```
x < 0 ? -x : x
```

If x is negative, the expression evaluates to -x (a positive value); otherwise to x.

The syntax for writing a conditional expression is:

```
<expr1> ? <expr2> : <expr3>
```

```
/* File: mathutil.c - continued*/
double maxdbl(double x, double y)
{
    return (x > y ? x : y);
}
```

Figure 5.11: Function `maxdbl` Using a Conditional Expression

The first operand, <expr1>, is evaluated; if true, the result of the entire expression is the value of < expr2>. Otherwise, the result is the value of <expr3>. The conditional operator is a ternary operator since it requires three operands.

An `if` statement can always perform the task that a conditional expression does. Whether to use one or the other is a matter of choice and convenience. Figure 5.11 shows a function which returns the value of the larger of two `double` arguments.

The Comma Operator

The comma operator, `,`, provides a way to combine several expressions into a single expression. The syntax is:

<expression1> , <expression2>

The semantics are that <expression1> is evaluated first, followed by <expression2> with the value of the entire expression being that of <expression2>. These expressions may be arbitrary expressions, including another comma expression.

The comma expression is useful where the syntax of a statement requires a single expression, but we have several expressions to be evaluated, such as a `for` statement where several variables are used to control the loop. Here, the comma operator may be used to write multiple initialization and update expressions. As an example, we will use comma operators to write a function that computes and prints Fibonacci numbers. Fibonacci numbers are natural numbers in the sequence:

```
1, 1, 2, 3, 5, 8, 13, ...
```

Each number of the sequence is computed by adding the previous two numbers of the sequence. Thus, we must start with the first two numbers, which are both 1, then the next number is $1 + 1 = 2$, the next one is $1 + 2 = 3$, the next one is $2 + 3 = 5$, and so on.

We will write a driver, `main()`, which calls a function, `fib()`, to print the Fibonacci numbers. The function starts with two variables, which are initialized to the values of the first two numbers 1 and 1. Each new number is computed as a sum of the previous two until the limit is reached. Figure 5.12 shows the code.

```
/* File: fib.c
     Program computes and prints Fibonacci numbers less than a
     specified limit of 100.
*/
#include <stdio.h>
#define LIM 100
void fib(int lim);

main()
{
    printf("***Fibonacci Numbers***\n");
    printf("Limit is %d \n", LIM);
    fib(LIM);
}


/* Function computes and prints the Fibonacci numbers less than lim. */
void fib(int lim)
{    int i, j, n;

    printf("1\n1\n");    /* print the first two fib. numbers */
    for (i = 1, j = 1, n = 0; n < lim; i = j, j = n) {
        n = i + j;              /* compute the next fib. number */

        if (n < lim)
            printf("%d\n", n);  /* print the next fib. number */
    }
}
```

Figure 5.12: Revised Fibonacci

The function, `fib()`, prints Fibonacci numbers less than its argument, `lim`. It uses a `for` loop with comma expressions for the first and last expressions. The first expression initializes two variables, `i` and `j` to 1, with `i` assumed to be the first and `j` assumed to be the second number in the sequence. The variable, `n`, the next number in the sequence, is initialized to zero so that the loop condition may be tested the first time with some value of `n` less than `lim`. The sum of `i` and `j` is the next number in the sequence, `n`, which is computed and printed in the loop body. The variables are then updated to the new values, `j` assigned the value of `n` and `i` assigned the value of `j`. Thus, `i` and `j` always have the values of the last two Fibonacci numbers in the sequence. The process is repeated until `n` exceeds `lim`.

The output of the program is shown below:

```
***Fibonacci Numbers***
Limit is 100
1
1
2
3
5
8
13
21
34
55
89
```

The `sizeof` Operator

The exact amount of space reserved in memory for different data types depends on the implementation. Typically, a character is assigned 8 bits or one byte of space; integers are generally assigned 2 or 4 bytes of storage; `float` numbers usually require at least four bytes, and `double` at least eight bytes. Table 5.3 shows some typical examples for the HP9000, an HP_UX Unix system, and the IBM PC, a DOS environment. It is sometimes necessary to use the sizes of objects in expressions, and since the sizes are implementation dependent, to make our programs portable, we should not build the values into our programs as constants. For any implementation, size of an object can be easily determined by the use of the `sizeof` operator with syntax:

```
sizeof <expression>
```

The unary operator, `sizeof`, yields the size, in bytes, of the type of its operand. The operand may be an arbitrary expression, however, the expression is NOT evaluated; the `sizeof` expression simply evaluates to the number of bytes used for the *type* of the result. For example, the expression, `sizeof x`, evaluates to the size of `x` in bytes. Here is a code fragment using the `sizeof` operator:

Data types	HP9000 Bytes	IBM PC Bytes
char	1	1
int	4	2
short int	2	2
long int	4	4
float	4	4
double	8	8
long double	16	8

Table 5.3: Space allocation in Bytes for data types

```
int x;
double y;

printf("Size of x is %d bytes\n",  sizeof x);
printf("Size of x+y is %d bytes\n",  sizeof (x+y));
```

The first `printf()` statement will print the size (in bytes) of the `int` type object, `x`. The second will print the size of the value of the expression, `x+y`. As we saw earlier, this addition would be done in double precision and the result would be a `double`. Remember, the expression, `x+y` is not evaluated; only its size is used by the `sizeof` operator. Also remember that `sizeof` is an operator, like `+`; not a function call. It has a precedence and associativity like any other operator (shown in Table 5.1). That is why the parentheses are required in that second `printf()`, the precedence of `sizeof` is higher than `+`. Without the parentheses, the expression would be evaluated as:

```
(sizeof x) + y
```

It is also possible for the operand of `sizeof` to be a parenthesized type name, like a cast operator, rather than a variable name, for example:

```
sizeof (int)
sizeof (float)
sizeof (long int)
sizeof (unsigned long int)
```

We can easily write a program to determine the sizes of different types for the host implementation. The code is shown in Figure 5.13. A sample output for the HP9000 is:

```
***Sizeof operator***
```

```
/* File: size.c */
main()
{   int x;
    double y;

    printf("***Sizeof operator***\n\n");
    printf("Size of x is %d bytes\n", sizeof  x);
    printf("Size of x+y is %d bytes\n\n", sizeof (x+y));
    printf("Size of data types in bytes:\n");
    printf("Size of int type is %d\n", sizeof(int));
    printf("Size of long int is %d\n", sizeof(long int));
    printf("Size of short int is %d\n", sizeof(short int));
    printf("Size of unsigned int  is %d\n", sizeof(unsigned int));
    printf("Size of float is %d\n", sizeof(float));
    printf("Size of double is %d\n", sizeof(double));
}
```

Figure 5.13: Testing `sizeof` Operator

```
Size of x is 4 bytes
Size of x+y is 8 bytes

Size of data types in bytes:
Size of int type is 4
Size of long int is 4
Size of short int is 2
Size of unsigned int is 4
Size of float is 4
Size of double is 8
```

Whenever the size of a type is required in a program, the `sizeof` operator should be used rather than the actual size, since the actual value is implementation dependent. Such a use of the `sizeof` operator in a program ensures that the program will be portable from one type of computer to another.

5.5   Common Errors

1. A result may be outside the range of values possible for a given data type. Use a data type with greater range and/or precision.

2. Prototypes are not declared; instead, default integer type declaration is assumed for functions. If there is no prototype declaration for a function and if the argument in the function call is a `float`, it is converted to `double`. If the formal parameter in the function definition

is declared as a `float`, there is a possible mismatch. A `double` object passed as an argument might be accessed as a `float` resulting in a possible wrong value. The actual situation depends on the compiler. Here is an example:

```
/*   File: default.c
     Program illustrates problems with default declarations for functions.
*/
#include <stdio.h>
main()
{    float x;

     x = 3.0;
     printf("Truncated Square of %f = %d\n", x, trunc_square(x));
}

int trunc_square(float z)
{
     return (int) (z * z);
}
```

The function `trunc_square()` returns integer type and `main()` uses the default declaration for `trunc_square()`. The `float` argument, `x` in the function call in `main()` is converted to `double`. But `trunc_square()` declares a `float` formal parameter, `z`. An attempt will be made to access a `double` object as a `float`. The function may not access the correct value passed as an argument. Thus, it is always best to use function prototypes to avoid confusion.

3. An expression is written without consideration of precedence and associativity of the operators. For example,

```
while (x = scanf("%d", &n) != EOF)
     . . .
```

Wrong! The `scanf()` value is compared first with `EOF` and the result of the comparison is assigned to `x`. Using parentheses:

```
while ((x = scanf("%d", &n)) != EOF)
     . . .
```

`x` is assigned the value returned by `scanf()`, and the value of `x` is then compared with `EOF`. Examples where associativity must be considered include:

```
a = 10; b = 5; c = 20; d = 4;

a - b - c       is -15
a / b / c / d   is 0
a % d % b % c   is 2
```

4. Increment and decrement operators are used incorrectly. Remember that postfix implies increment/decrement after evaluation and prefix implies increment/decrement before evaluation.

5.6   Summary

In this chapter we have tied up some loose ends and formalized some of the concepts from previous chapters. We have seen how the finite number of bits available to represent numbers limits the range and precision of the numbers stored in the computer. We have introduced additional data types which can extend the range and increase precision as needed for some applications. We have discussed the data types `void` (when no value is expected) and `enum` (for improving program readability). We have also shown how user defined names for data types can be defined using `typedef` with syntax:

    typedef <existing-type-specifier> <new-type-specifier>;

We have extended our available control constructs by introducing two variations on the looping constructs provided in the language: the `for` statement and the `do...while` statement, with syntax:

```
                                                       <expr1>;
                                                       while (<expr2>) {
        for (<expr1>; <expr2>; <expr3>) <statement>  equivalent to      <statement>
                                                             <expr3>;
                                                       }
```

and

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

We have also described how expressions are evaluated, including the determination of the type of the result and the order of applying operators, giving the full precedence and associativity table for all C operators (Table 5.1). We have described some new operators, such as the increment/decrement operators:

    ++ <Lvalue>
    -- <Lvalue>
    <Lvalue> ++
    <Lvalue> --

composite assignment operators:

    <Lvalue> <op>= <expression>

the conditional expression:

<expr1> ? <expr2> : <expr3>

the comma operator:

<expression1> , <expression2>

and the `sizeof` operator:

sizeof <expression>

Other operators in the table such as the indirection, array subscripting, structure accessing, and bitwise operators will be described in later chapters.

5.7   Exercises

1. If x is 100 and z is 200, what is the output of the following:

   ```
   if (z = x)
        printf("z = %d, x = %d\n", z, x);
   ```

2. With the following declarations:

   ```
   int a = 10, b = 15, c = 25;
   float x = 15;
   double y = 30;
   long int m = 25L;
   ```

   What are the values and types of the following expressions:

   ```
   a + b / c * x;
   a + b / x * c;
   a + b / y * c;
   a + b / m * c;

   x + a / b;
   x + (int) a / b;
   ```

3. Evaluate the expressions following the declarations:

   ```
   int x, y, z;
   float u, v, w;

   x = 10; y= 20; z = 30;

   x = z / y + y;
   x = x / y / z;
   x = x % y % z
   ```

4. Evaluate the expressions:

   ```
   int x, y, z;
   float u, v, w;
   x = 10; y= 20; z = 30;
   u = 5.0; v = 10.0; w = 30.0;

   x = w / y + y;
   u = z / y + y;
   u = w / y + y;
   u = x / y / w + u / v;
   ```

5. What is the output of the following program?

```
#define PRHAPS
#define TWICEZ z + z

main()
{    int w, x, y, z;
     float a, b, c;
     w = 16; x = 5; y = 15; z = 8;
     a = 1.0; b = 2.0; c = 4.0;

     #ifdef PRHAPS
          x = 15;
          y = 5;
     #endif

     printf("(a). %d %d\n", x, y);
     printf("(b). %d\n", TWICEZ * 2);
     printf("(c). %f %f\n", w / z * a + c, z / w * b + c);
     printf("(d). %d\n", z % y % x);
}
```

6. What will be the output in the following cases:

(a)
```
#define SWAP(x, y)   int temp; temp = x; x = y; y = temp
main()
{    int x1 = 10, x2 = 20;

     SWAP(x1, x2);
     printf("x1 = %d, x2 = %d\n", x1, x2);
}
```

(b)
```
#define SWAP(x, y)   {int temp; temp = x; x = y; y = temp;}
main()
{    int x1 = 10, x2 = 20;

     SWAP(x1, x2);
     printf("x1 = %d, x2 = %d\n", x1, x2);
}
```

(c)
```
#define SWAP(x, y)   int temp; temp = x; x = y; y = temp
main()
{    int x1 = 10, x2 = 20;

     printf("Swapping Values\n");
     SWAP(x1, x2);
     printf("x1 = %d, x2 = %d\n", x1, x2);
}
```

7. Write a `while` and a `do...while` loop to read and echo long integers until end of file. Allow for the possibility that the first input is an end of file.

8. Write a for loop to print out squares of integers in the sequence 5, 10, 15, 20, 25, etc. until 100.

9. Given the following declarations:

```
int x = 100, y;
```

What are the values of x and y after each of the following expressions is evaluated (the expressions are evaluated in sequence)?

```
y = x++;
y = ++x;
y = --x;
y = x--;
```

10. What are the values of the following expressions considered sequentially:

```
x = 100; y = 200;
y = y++ - ++x;
y = ++y - x++;
y = ++y * 2;
y = 2 * x++;
```

11. Evaluate the following:

```
x = 100; y = 200;
y += 2 * x++;
y -= 2 * --x;
y += x;
```

12. Evaluate the following:

```
x = 100; y = 200; z = 25;
z = y > x ? x : y;
z = (z >= x && z >= y) ? z - x * y : z + x * y;
```

5.8   Problems

1. Write a program to calculate the roots of a quadratic equation:

$$a * x^2 + b * x + c = 0$$

   The program should repeatedly read the set of coefficients a, b, and c. For each set, calculate the roots if and only if $b * b$ is not less than $4 * a * c$. Otherwise, write a message that the roots are not real and proceed to the next set of coefficients. The two roots of a quadratic are:

$$x_1 = \frac{-b + \sqrt{b^2 - 4 * a * c}}{2 * a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4 * a * c}}{2 * a}$$

   Use the `sq_root()` function defined in the chapter.

2. Write a function to find `exp(x)` whose value is given by the Taylor series:

$$1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

   where $n!$ is n factorial. Write and use a function, `power(x, n)`, which returns the nth power of `x`, where `n` is an integer. Use a function, `fact()`, to compute the factorial. Write a driver that reads input values of `x`, and finds `exp(x)`. Use as many terms as needed to make values before and after an additional term very close.

3. Write a function to evaluate `sin(x)` using the expansion shown below. Use it in a program to find the sine of values read until end of file.

$$sin(x) = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

4. Write a function, `cos(x)`, using the expansion below and use it in a program to find the cosine of values read until `EOF`.

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

5. What are the limitations on the accuracy of the above expansions?

6. Write a function that returns the number of ways that `r` items can be taken together out of `n` items. The value of combination is:

$$comb(n, r) = \frac{n!}{(n - r)! * r!}$$

   Use long integers for factorials.

7. Extend the range of possible values for Problem 6 by cancelling out common factors in numerator and denominator.

8. Write a program that uses Newton's method to find the roots of the equation:

$$f(x) = x^2 + 5 * x + 6 = 0$$

Newton's method uses successive approximations. Start with a guess value for root. The improved value of root is given by:

$$newroot = root - \frac{f(root)}{f'(root)}$$

where $f(root)$ is the value of the function when x equals $root$, and $f'(root)$ is the value of the function below when x equals $root$:

$$f'(x) = 2 * x + 5$$

9. Write a program that finds the approximate value of an integral of a function whose four sample values s_1, s_2, s_3, s_4 are specified at time instants t_1, $t_1 + h$, $t_1 + 2 * h$, $t_1 + 3 * h$. The user should be asked for the value of the interval size, h, and starting instant, t_1. The approximate value of an integral from t_1 to $t_1 + 4 * h$ is the sum of the area under each rectangle made up of the sample value and the inter-sample distance, i.e.:

$$s_1 * h + s_2 * h + s_3 * h + s_4 * h$$

10. Write a program that reads in the coefficients and the right hand side values for two linear simultaneous equations. Solve the equations for the unknowns and print the solution values. The equations are:

$$a_{(1,1)} * x_1 + a_{(1,2)} * x_2 = c_1$$

$$a_{(2,1)} * x_1 + a_{(2,2)} * x_2 = c_2$$

where $a_{(1,1)}$, $a_{(1,2)}$, c_1, $a_{(2,1)}$, $a_{(2,2)}$, and c_2 are the coefficients to be read, and x_1 and x_2 are the unknowns. To solve the equations, multiply the first equation coefficients and right hand side by $-\frac{a_{(2,1)}}{a_{(1,1)}}$ and add the corresponding values to those of the second equation. The new, modified value of $a_{(2,1)}$ will be zero, so the second equation can be solved for x_2, and, substituting the value of x_2 in the first equation, solve for x_1.

11. Given coefficients and the right hand side of two simultaneous equations, verify if a given set of values for x_1 and x_2 is correct. If the left hand side and the right hand side are within a small error margin the solution is assumed to be correct. Let the margin of error be a specifiable value with an assumed default value.

12. Write a menu-driven program to solve and verify two linear equations as per Problems 10 and 11. Allow the following commands: get data, display data, solve equations, display solution, verify solution, help, and quit.

13. Write a program to determine the current and the power consumed in an electrical resistor (load) of 10000 ohms if it is connected to a battery of 12 volts. Power consumed in a resistor is V^2/R, where V is the volts across the resistor and R is the resistor value in ohms. The current in a resistor is given by V/R.

14. Use `for` loops to write a program that finds all prime numbers less than a specified value.

15. Use `do...while` loops to write Problem 14.

16. Write a program that reads a year, a month, and a day of the month. It then determines the number of the day in the year. (Use the definition of a leap year given in Problem 3.6). Use enumeration type for the months, and a `switch` statement which uses the number of days in the year prior to the first of each month.

17. Modify Problem 16 so the program reads the day of the week on the first of January and determines the day of the week for the specified date.

18. Write a program to read the current date in the order: year, month, and day of the month. The program then prints the date in words: Today is the nth day of Month of the year Year. Example:

    ```
    Today is the 24th day of December of the year 2000.
    ```

19. If the GCD of two numbers, m and n is 1, they have no common divisor. Write a program to find all pairs of numbers, in the range 2 to 20, that have no common divisors. (Refer to Problem 3.12 for the definition of GCD).

20. A rational number is maintained as a ratio of two integers, e.g. 20/23, 35/46, etc. Rational number arithmetic adds, subtracts, multiplies and divides two rational numbers. Write a program that repeatedly reads and adds two rational numbers. The program should print the result in each case as a rational number.

21. Write a function to subtract two rational numbers.

22. Write a function to multiply two rational numbers.

23. Write a function to divide two rational numbers.

24. Write a function to reduce a rational number. A reduced rational number is one in which all common factors in the numerator and the denominator have been cancelled out. For example, 20/30 is reduce to 2/3, 24/18 is reduced to 4/3, and so forth. The GCD can be used to reduce a rational number.

25. Modify the rational numbers programs in Problems 20 through 24 so the result is first reduced before it is printed.

Chapter 6

Pointers

In the preceding chapters, our programs have been written to access objects directly, i.e. using the variable names. We have postponed until now a discussion of the concept of **indirect access**, i.e. access of objects using their address. As we have seen, variables local to a function may be accessed using their name *only* within that function. When arguments are passed to another function, only the values are passed, and the *called* function may use these values, but cannot affect the variable cells in the *calling* function. Sometimes, however, a function needs to have direct access to the cells in another function. This can be done in C through indirect access, using the address of the cell, called a **pointer**.

In this chapter, we will introduce the concepts of indirect access, pointer types, and dereferenced pointer variables. We will use these concepts to write functions that indirectly access objects in a calling function.

6.1 What is a Pointer?

Frequently, a *called* function needs to make changes to objects declared in the *calling* function. For example, the function, `scanf()`, needs to access objects in the calling function to store the data read and converted into an object defined there. Therefore, we supply `scanf()` with the *address* of objects rather than their values. Here, we will see how any function can indirectly access an object by its address.

Another common use of pointers is to write functions that "return" more than one value. As we have seen, every function in C returns a value as the value of the function; however, if a function's *meaning* includes the return of several pieces of information, this single return value is not sufficient. In these cases, we can have the function return multiple data values indirectly, using pointers.

6.1.1   Data vs Address

Before we discuss passing pointers and indirectly accessing data between functions, let us look at how we can declare pointer variables and access data using them. Consider the following simple program:

```
main()
{    int x;
     int iptr;

     printf("***Testing Pointer Variables***\n");
     x = 10;
     iptr = &x;
     printf("%d\n",iptr);

}
```

We have declared two integers, `x`, intended to hold an integer value, and `iptr` which is intended to hold a pointer to an integer, i.e. and address of an integer. We then assign a value to `x`, and the address of `x` to the variable `iptr` using the `&` (address of) operator. The address of a variable is simply the byte address of the cell which was allocated by the declaration. An address is an integer (actually and unsigned integer) so may be stored in an `int` type variable. The situation is shown in Figure 6.1a). When we compile and execute this program the result is:

```
***Testing Pointer Variables***
1000
```

What if we had wanted to print the value of the cell pointed to by `iptr` and not the value of `iptr` itself? The **indirection operator**, `*`, accesses the object pointed to by its operand. In our example, the value of `iptr` is 1000 which is an address of some object; i.e. `iptr` points to *some* object located at address 1000. So we should be able to access that object with an expression like:

```
*iptr
```

However, there is no way to know how many bytes to access at address 1000, nor how to interpret the data, unless the type of object at address 1000 is known: is it an `int`? a `float`? a `char`? etc. In order for the compiler to know how to access an object indirectly, it must know the type of that object. We specify the type of object to access by indicating to the compiler the type of objects a pointer refers to when we declare the pointer. So, in our example, we should declare the variable, `iptr` as a "pointer to an integer" as follows:

```
int *iptr;
```

or,

```
    int * iptr;
```

(white space may separate the operator, *, and the variable name, iptr). The declaration specifies
a variable, iptr, of type int *, i.e. integer pointer (the type is read directly from the declaration).
So, int * is the type of iptr, and int is the type of *iptr — the thing it points to. This statement
declares an integer pointer variable, iptr, and allocates memory for a pointer variable. Similarly,
we can declare float pointers or character pointers:

```
    float * pa, * pb;
    char  * pc;
```

These statements declare variables, pa and pb, which can point to float type objects, and pc
which can point to a char type object. All pointer variables store addresses, which are unsigned
integers, and so need the same amount of memory space regardless of the pointer types.

Since the compiler now knows that iptr points to an integer object, it can access the object
correctly. Our simple program becomes:

```
    main()
    {   int x;
        int *iptr;

        printf("***Testing Pointer Variables***\n");
        x = 10;
        iptr = &x;
        printf("Address %d holds value %d\n",iptr,*iptr);

    }
```

which produces the output:

```
    ***Testing Pointer Variables***
    Address 1000 holds value 10
```

We are generally not interested in the *value* of the pointer variable itself; it may even be different
each time a program is run. Instead, we are interested in the cell the pointer is pointing to, so
we indicate the *value* of a pointer variable in diagrams and program traces using an arrow (←) as
shown in Figure 6.1b.

In summary, the address of an object is called a **pointer** to that object since the address tells
one where to go in order to access the object. The address by itself does not provide sufficient

main()

int x

10

int iptr

1000

a)

main()

int x

10

int *iptr
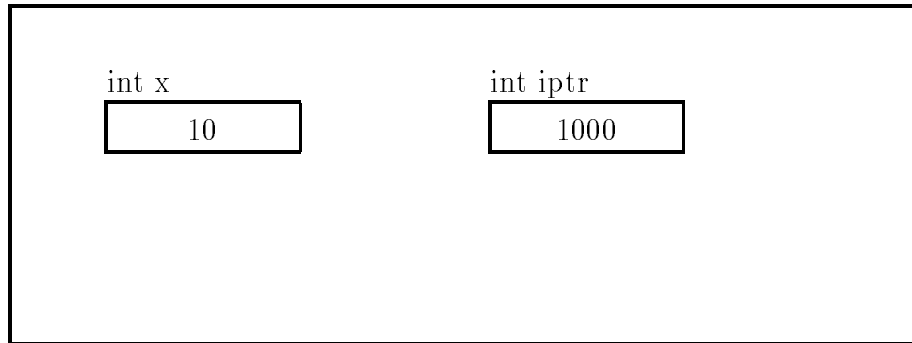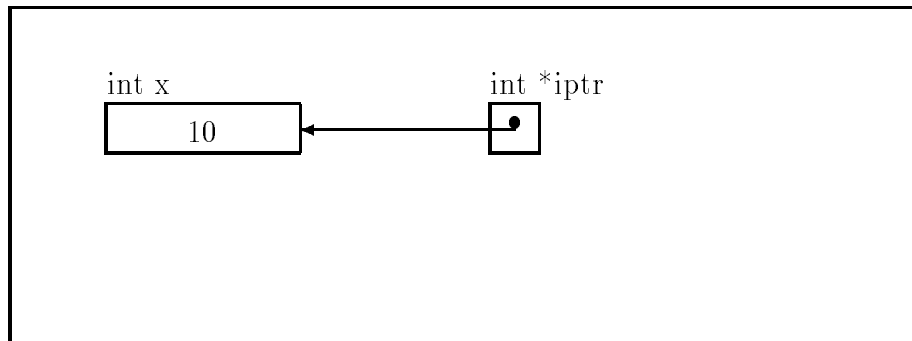
Figure 6.1: Declaring Pointer Variables

b)

information to access an object; we must know what type of object the address is pointing to. If the pointer (address) value and the data type of the object that it points to are both known, then it is possible to access the object correctly. In other words, pointers must be specified to be `int` pointers, pointing to an integer type object, `float` pointers, pointing to a floating point type object, `char` pointers, etc.

6.1.2 Indirect Access of Values

The indirection operator, `*`, accesses an object of a specified type at an address. Accessing an object by its address is called **indirect access**. Thus, `*iptr` indirectly accesses the object that `iptr` points to, i.e. `*iptr` *accesses* `x`. The indirection operator is also called the **contents of operator** or the **dereference operator**. Applying the indirection operator to a pointer variable is referred to as **dereferencing** the pointer variable, i.e. `*iptr` *dereferences* `iptr`. The address of operator, `&`, is used to get the address of an object. We have already used it in calls to `scanf()`. We can also use it to assign a value to a pointer variable.

Let us consider some examples using the following declarations:

```
int x, z;
float y;
char ch, * pch;
int * pi, *pi2;
float * pf;
```

When these declarations are encountered, memory cells are allocated for these variables at some addresses as shown in Figure 6.2. Variables `x` and `z` are `int` types, `y` is `float`, and `ch` is `char`. Pointer variables `pi` and `pi2` are variables that can point to integers, `pf` is a `float` pointer, and `pch` is a character pointer. Note that the initial values of all variables, including pointer variables, are unknown. Just as we must initialize `int` and `float` variables, we must also initialize pointer variables. Here are some examples:

```
x = 100;
y = 20.0
z = 50;

pi = &x;        /* pi points to x */
pi2 = &z;       /* pi2 points to z */
pch = &ch;      /* pch points to ch */
```

The result of executing these statements is shown in Figure 6.3: `pi` points to the cell for the variable `x`, `pi2` points to `z`, `pch` points to `ch`, and `pf` still contains garbage. Remember, the *value* of a pointer variable is stored as an address in the cell; however, we do not need to be concerned with the value itself. Instead, our figure simply shows what the initialized pointer variables point

main()
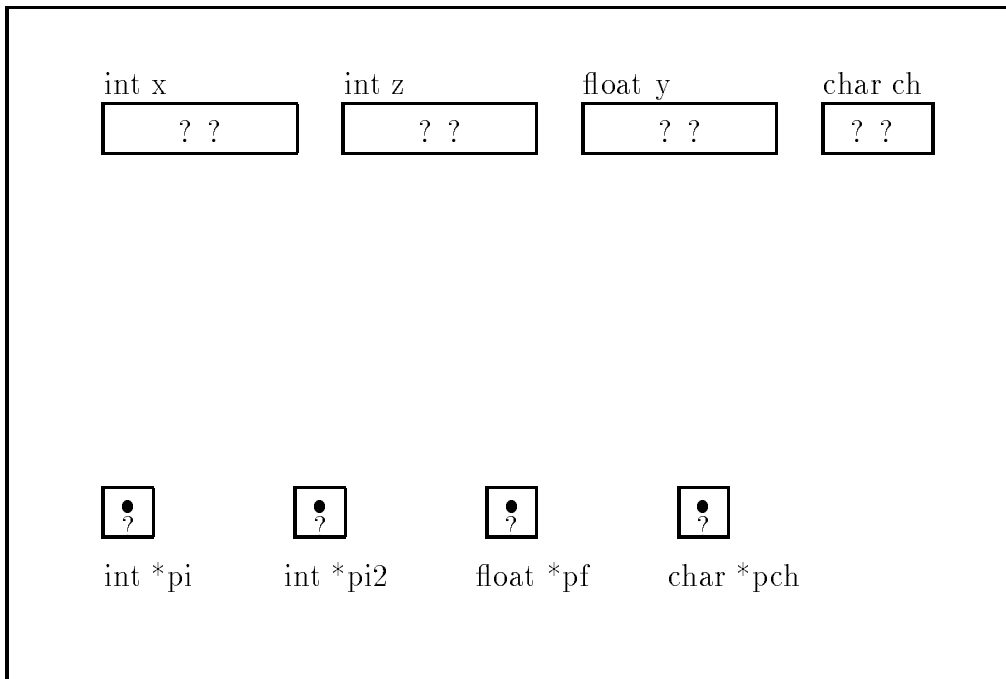
Figure 6.2: Declaration of Pointer Variables

main()
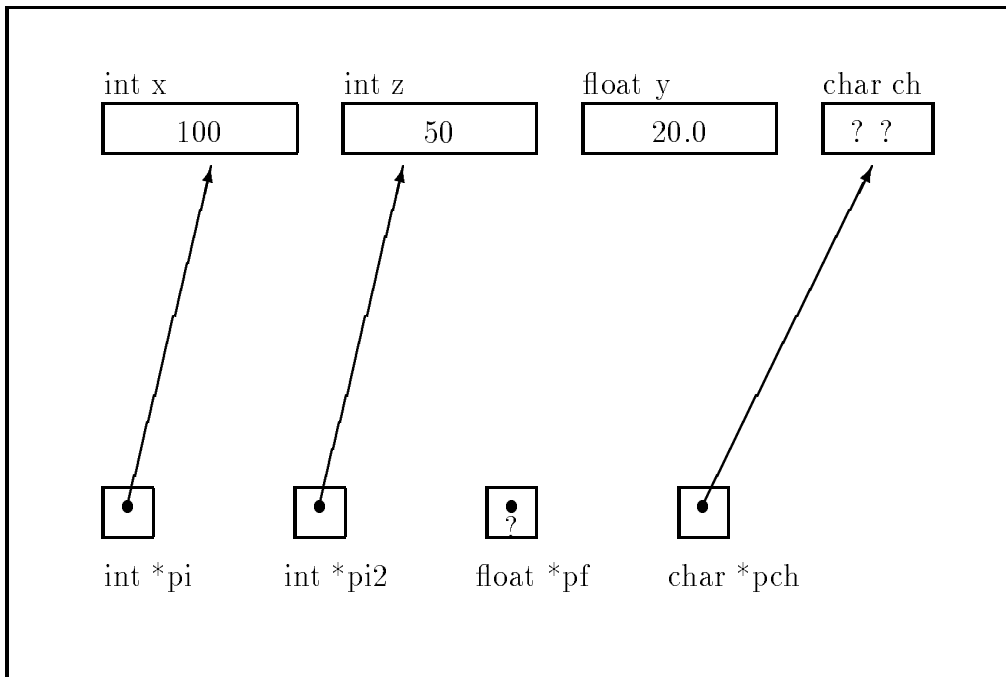
Figure 6.3: Assignments of pointers

main()
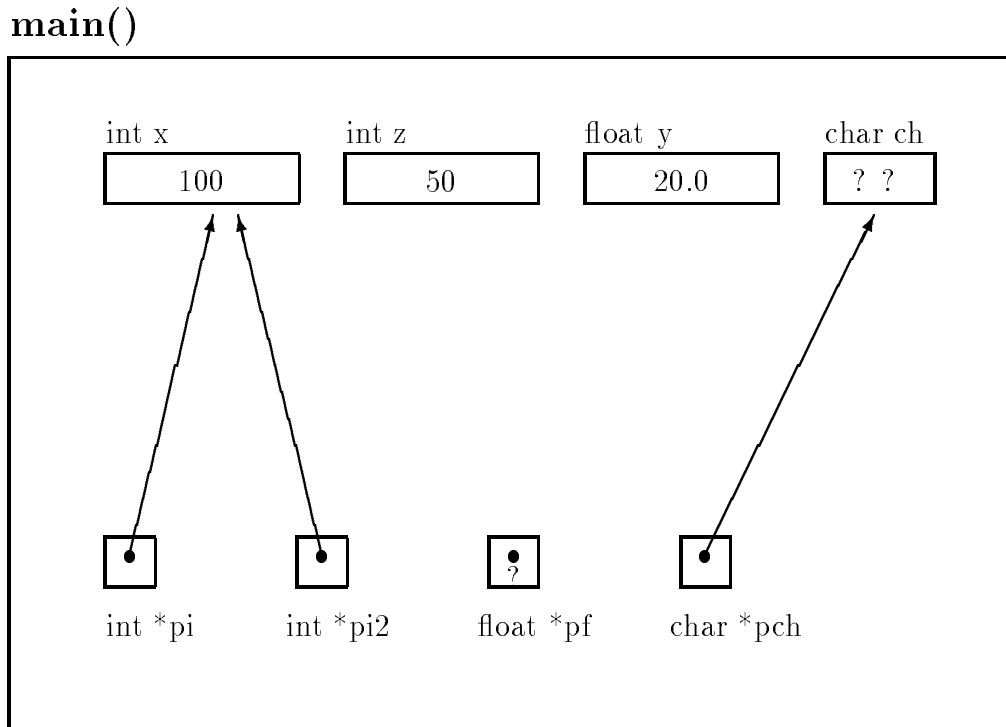
Figure 6.4: Effect of Pointer to Pointer Assignment — Statement 1.

to. These initialized pointers may now be used to indirectly access the objects they point to, or they be may be changed by new assignments. Here are some examples of statements and how they change things for the above memory organization. (The statements are numbered in order to reference them; the numbers are not part of the code).

```
1:   pi2 = pi;            /* pi2 points to where pi points          */
                          /* i.e. pi2 ==> x                         */
2:   pi = &z;             /* pi now points to z, pi2 still points to x */
                          /* i.e. pi ==> z, pi2 ==> x               */
3:   *pi = *pi2;          /* z = x, i.e, z = 100                    */
4:   *pi = 200;           /* z = 200, x is unchanged                */
5:   *pi2 = *pi2 + 200;   /* x = 300, z is unchanged                */
```

Statement 1: Assigns value of `pi` to `pi2`, so `pi2` now also points to `x` (see Figure 6.4). Since both of the variables are type `int *` this assignment is allowed.

Statement 2: Makes `pi` point to `z` (see Figure 6.5). The expression `&z` evaluates to the address of `z`; i.e. an `int` pointer.

Statement 3: Since `pi2` points to `x`, the value of the right hand side, `*pi2`, dereferences the pointer and evaluates to the value in the cell, i.e. 100. This value is assigned to the object accessed by the left hand side, `*pi`, i.e. the place pointed to by `pi` or the

main()
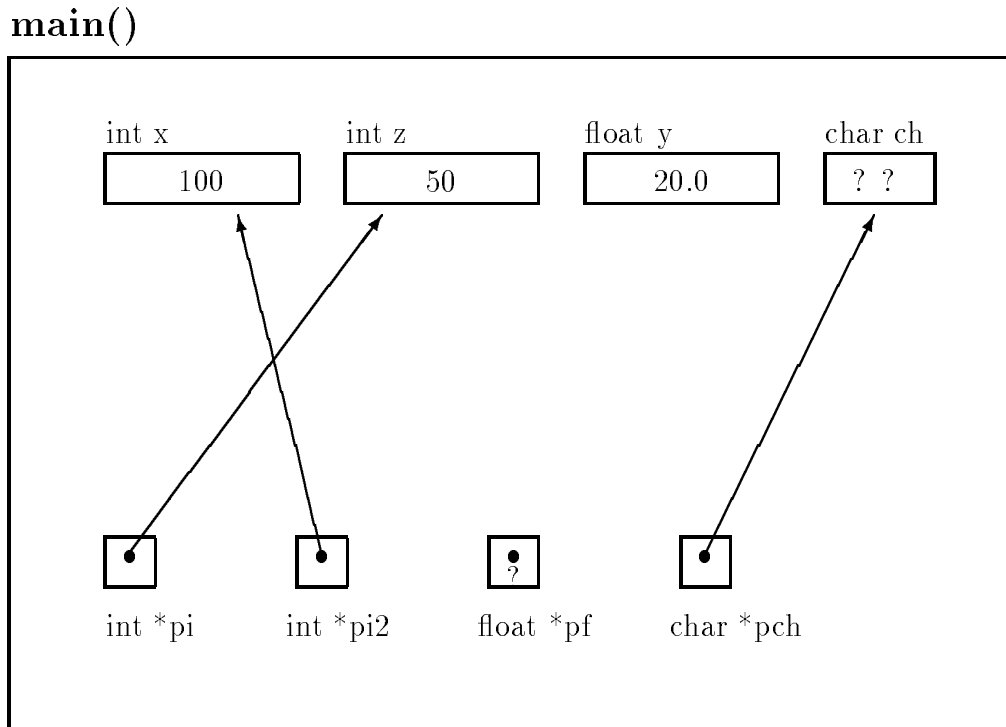
Figure 6.5: Effect of Pointer Reassignment — Statement 2.

object `z` (see Figure 6.6). This has the same effect as the assignment `z = x`. Note, we have used a dereferenced pointer variable as the Lvalue on the left hand side of an assignment operator. The semantics is to access the object indirectly and store the value of the expression on the right hand side.

Statement 4:   The value, 200, is assigned to `*pi`, i.e. `z` (see Figure 6.7). Again, we have used an indirect access for the Lvalue of the assignment.

Statement 5:   The right hand side evaluates to 300, since 200 is added to `*pi2`; so 300 is assigned to `*pi2`, i.e. `x` (see Figure 6.8). Again, we have used an indirect access on both the left and right hand sides.

We see that the left hand side of an assignment operator, the Lvalue, can be a reference to an object either by direct access (i.e. a variable name) or by indirect access (i.e. a dereferenced pointer variable). Also notice that we were very careful about the *type* of the objects on the left and right hand side of the assignment operators. We have assigned an integer value to a cell pointed to by an integer pointer, and when assigning pointers, we have assigned an integer pointer to a cell declared as an `int *`. An assignment statement such as:

```
pi = x;
```

main()
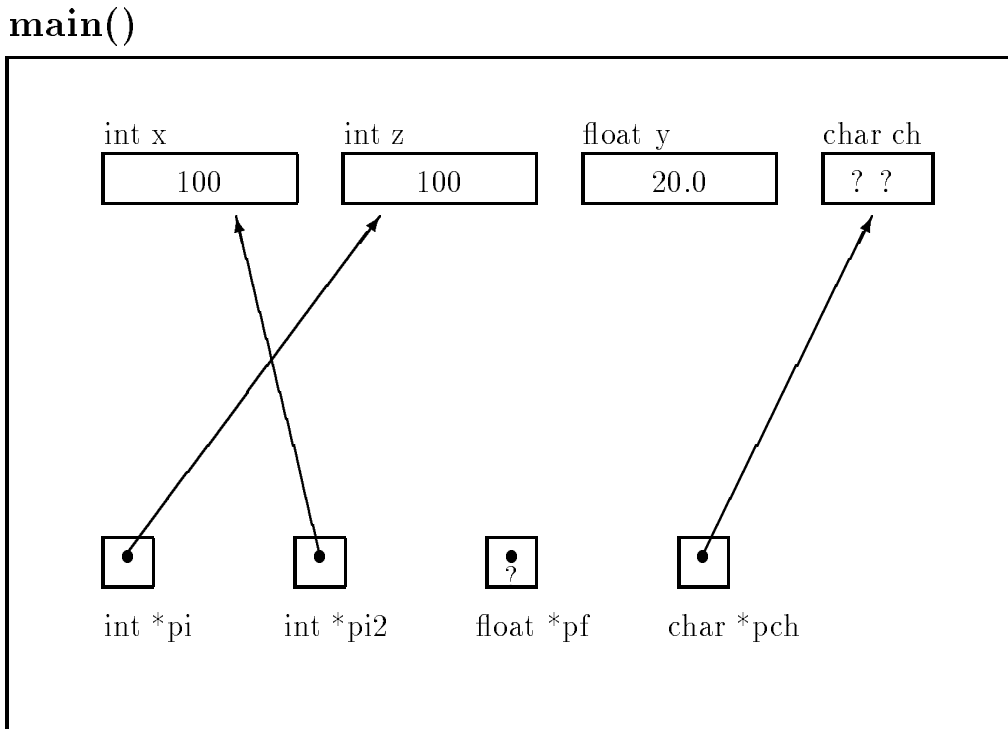
Figure 6.6: Effect of Indirect Pointer Access and Assignment — Statement 3

main()
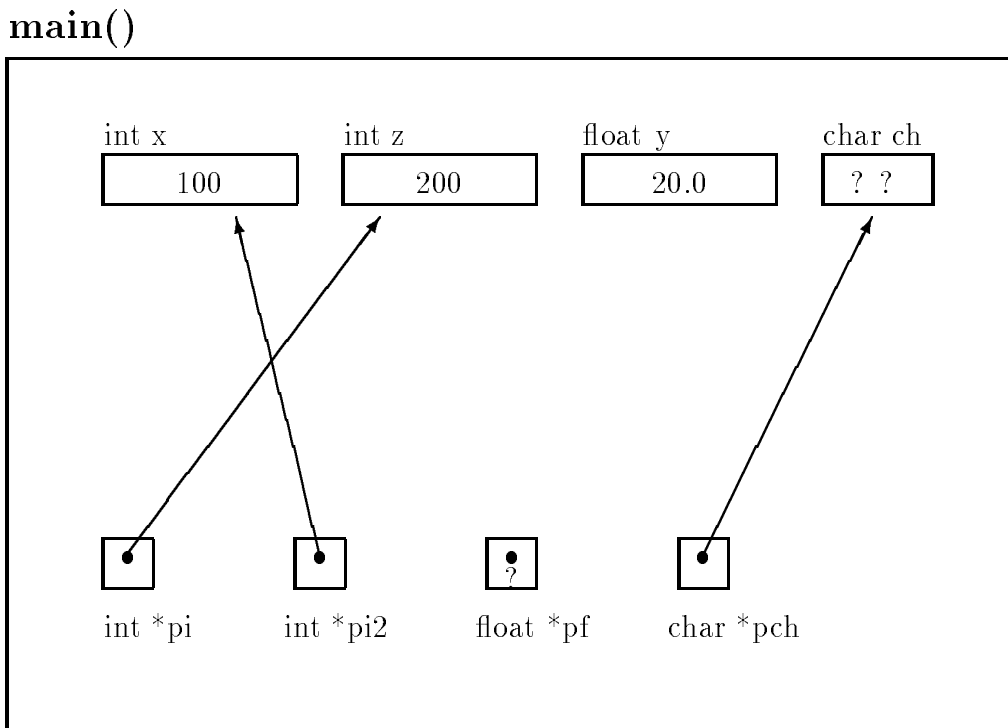
Figure 6.7: Effect of Indirect Assignment — Statement 4

main()
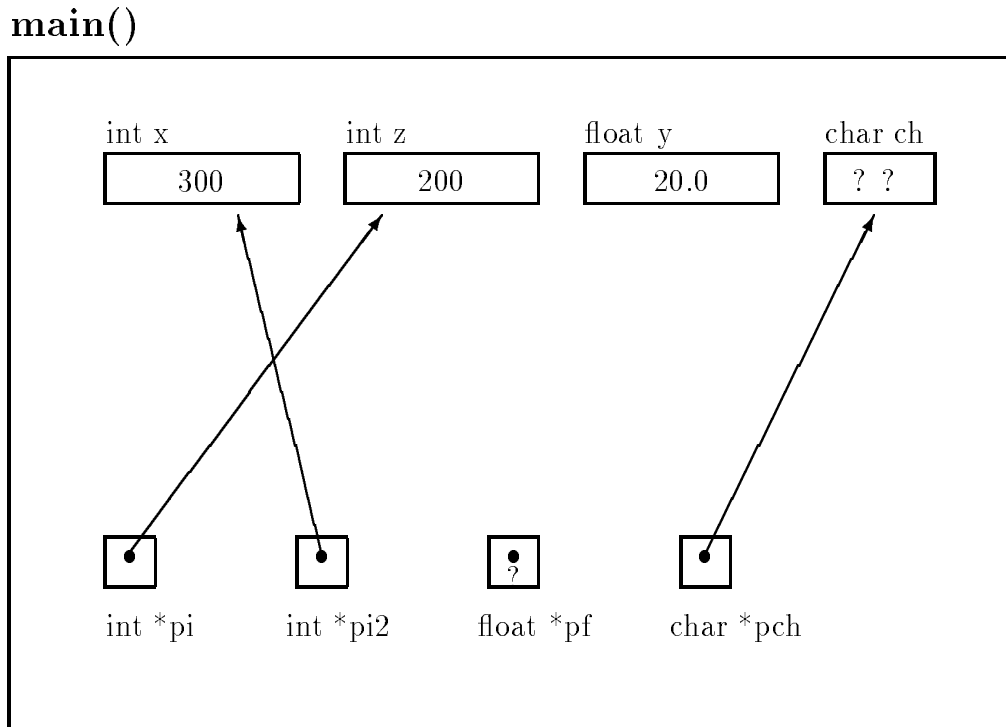
Figure 6.8: Effect of Indirect Pointer Access and Assignment — Statement 5

is a legal statement in C: assigning an integer value to a pointer cell. However, the effect may not be as we would expect. The value of **x** will be placed in the pointer cell, **pi**, and subsequent dereferencing of **pi**, (***pi**), will use that value as a pointer (an address) to find the cell to indirectly access. This is almost *never* what we intend to do in this statement. Most C compilers will generate a *warning* at compile time stating that an illegal integer-pointer combination in an assignment was encountered to indicate that something is *possibly* wrong here. A warning is not an error; it does not prevent the compiler from generating a functional object file. However, it is an indication that the statement may not be what the programmer intended. Such a statement is probably correctly written as:

    *pi = x;   or   pi = &x;

which assign a value to the cell pointed to by **pi** or to assign an address to **pi** itself, respectively. (In the RARE instance where such an assignment of an integer to a pointer cell is intended, the syntax:

    pi = (int *)x;

i.e. casting the integer to an integer pointer, should be used).

Likewise, an attempt to use the uninitialized variable, **pf** will be a disaster. Suppose we write:

```
printf("%f\n", *pf);
```

The value of pf is garbage so *pf will attempt to access the garbage address for a float object. The garbage value of pf may be an invalid memory address, in which case, the program will be aborted due to a *memory fault*; a run time error. This is bad news; however, we may be even more unfortunate if the value in pf is a valid memory address. In this case, we would access a value from some unknown place in memory. The situation is even worse when an uninitialized pointer is used indirectly as an Lvalue:

```
*pf = 3.5;
```

Since we do not know where pf is pointing, if it happens to be a legal address, we have just placed the value, 3.5, in some unknown location in memory, possible a cell belonging to a variable in another part of the program. Finding this type of bug is very difficult. The lesson here is that care should be taken when using pointers, particularly ensuring that pointers are properly initialized.

On the other hand, the character variable, ch, is not initialized, but the pointer variable, pch is initialized to point to ch so the expression, *pch, will access the object, ch, correctly. If the value of *pch is accessed, it will be garbage; but a value can be stored in *pch correctly.

With proper care, the value of an initialized pointer variable (the address of some object) allows us to indirectly access the object by dereferencing the pointer variable. An example program, shown in Figure 6.9, illustrates the *value of a pointer variable* and the *value of the object indirectly accessed* by it.

Figure 6.10 shows program trace graphically. The program first declares an int and an int * variables (Figure 6.10a)). The first printf() statement prints the program title followed by the initialization of i1 and iptr (Figure 6.10b)). The next printf() gives the hexadecimal value of iptr, which is the address of i1. The next statement prints the value of the same object indirectly accessed by *iptr and directly accessed by i1. Then, the value of *iptr is changed (Figure 6.10c)); and the last statement prints the changed value of the object, accessed first indirectly and then directly.

The output for a sample run is:

```
Pointers:  Direct and Indirect Access

iptr = 65490
*iptr = 10, i1 = 10
*iptr = 100, i1 = 100
```

```
/*    File: access.c
      This program prints out the values of pointers and values of
      dereferenced pointer variables.
*/
#include <stdio.h>
main()
{     int *iptr,      /* integer pointer */
            i1;

      printf("Pointers: Direct and Indirect Access\n\n");
      /* initializations */
      i1 = 10;
      iptr = &i1;     /* iptr points to the object whose name is i1 */

      /* print value of iptr, i.e., address of i1 */
      printf("iptr = %u\n", iptr);
      /* print value of the object accessed indirectly and directly */
      printf("*iptr = %d,   i1 = %d\n", *iptr, i1);

      *iptr = *iptr * 10;       /* value of *iptr changed */
      /* print values of the object again */
      printf("*iptr = %d,   i1 = %d\n", *iptr, i1);
}
```

Figure 6.9: Example Code with Direct and Indirect Access

main()

int i1

? ?

•

int *iptr

main()

int i1

10

•

int *iptr

main()

int i1

100

•

int *iptr

a)

b)

c)
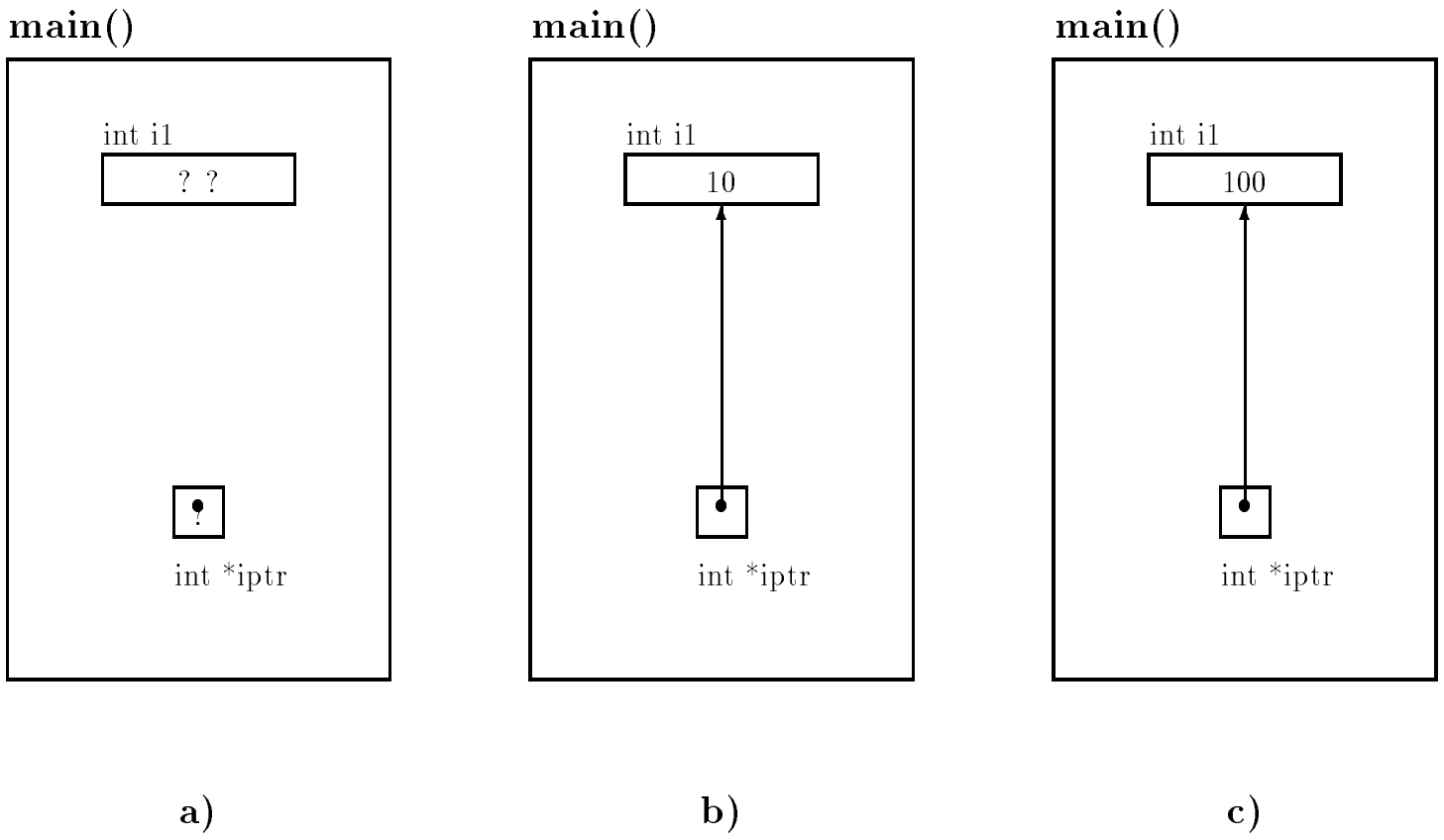
Figure 6.10: Trace for Direct and Indirect Access

6.2   Passing Pointers to Functions

As we have seen, in C, arguments are passed to functions *by value*; i.e. only the *values* of argument expressions are passed to the called function. Some programming languages allow arguments passed *by reference*, which allows the called function to make changes in argument objects. C allows only call by value, not call by reference; however, if a called function is to change the value of an object defined in the calling function, it can be passed a value which is a *pointer* to the object. The called function can then dereference the pointer to access the object indirectly. We have also seen that a C function can return a single value as the value of the function. However, by indirect access, a called function can effectively "return" several values. Only one value is actually returned as the value of the function, all other values may be indirectly stored in objects in the calling function. This use of pointer variables is one of the most common in C. Let us look at some simple examples that use indirect access.

6.2.1   Indirectly Incrementing a Variable

We will first write a program which uses a function to increment the value of an object defined in `main()`. As explained above, the called function must indirectly access the object defined in `main()`, i.e. it must use a pointer to access the desired object. Therefore, the calling function must pass an argument which is a pointer to the object which the called function can indirectly access.

Figure 6.11 shows the code for the program and the program trace is shown graphically in Figure 6.12. The function, `main()` declares a single integer variable and initializes it to 7 (see Figure 6.12a)). When `main()` calls `indirect_incr()`, it passes the pointer, `&x` (the address of `x`). The formal parameter, `p`, is defined in `indirect_incr()` as a pointer variable of type `int *`. When `indirect_incr()` is called, the variable, `p` gets the value of a pointer the the cell named `x` in `main()` (see Figure 6.12b)). The function, `indirect_incr()`, indirectly accesses the object pointed to by `p`, i.e. the `int` object, `x`, defined in `main()`. The assignment statement indirectly accesses the value, 7, in this cell, and increments it to 8, storing it indirectly in the cell, `x`, in `main()` (see Figure 6.12c)).

Sample Session:

```
***Indirect Access***
Original value of x is 7
The value of x is 8
```

6.2.2   Computing the Square and Cube

Sometimes, whether a value should be returned as the value of a called function or indirectly stored in an object is a matter of choice. For example, consider a function which is required to "return"

```
/*    File: indincr.c
      Program illustrates indirect access
      to x by a function indirect_incr().
      Function increments x by 1.
*/
#include <stdio.h>
void indirect_incr(int * p);


main()
{    int x;

     printf("***Indirect Access***\n");
     x = 7;
     printf("Original value of x is %d\n", x);
     indirect_incr(&x);

     printf("The value of x is %d\n", x);
}



/*    Function indirectly accesses object in calling function.  */
void indirect_incr(int * p)
{
     *p = *p + 1;
}
```

Figure 6.11: Code for Indirect Access by a Function

main()

int x

7

main()

int x

7

indirect_incr(int * •)

p

main()

int x

8

indirect_incr(int * •)

p

a)   b)   c)
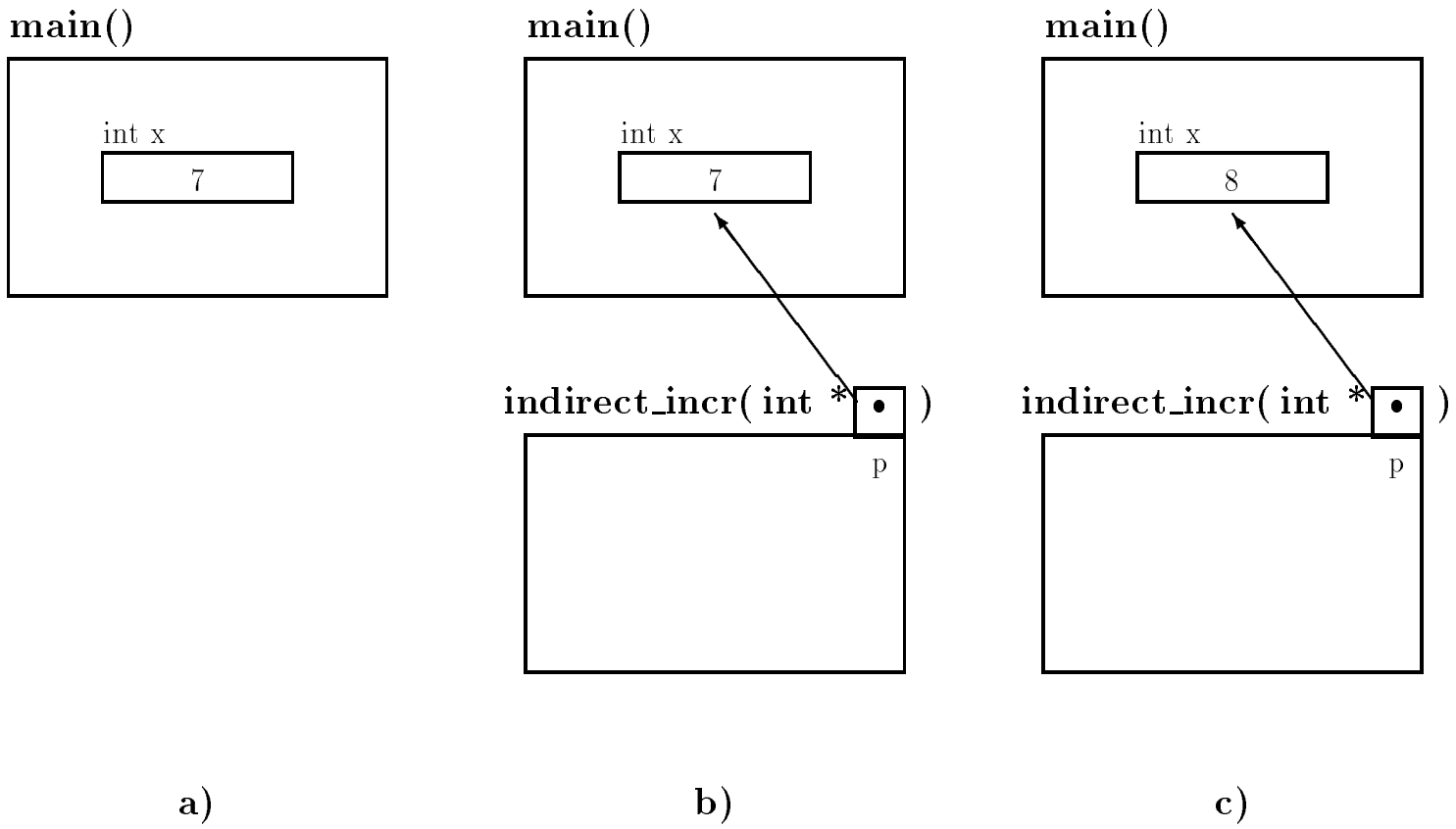
Figure 6.12: Trace for Indirect Access by a Function

```
/*   File: sqcube.c
     Program uses a function that returns a square of its argument and
     indirectly stores the cube.
*/
#include <stdio.h>
double sqcube(double x, double * pcube);

main()
{    double x, square, cube;

     printf("***Directly and Indirectly Returned Values***\n");
     x = 3;

     square = sqcube(x, &cube);
     printf("x = %f, square = %f, cube = %f\n",
                x, square, cube);
}

/* Function return square of x, and indirectly stores cube of x */
double sqcube(double x, double * pcube)
{
     *pcube = x * x * x;
     return (x * x);
}
```

Figure 6.13: Code for Indirectly Returned Values

two values to the calling function. We know that only one value can be returned as the value of the function, so we can decide to write the function with one of the two values *formally* returned by a `return` statement, and the other value stored, by indirect access, in an object defined in the calling function. The two values are "returned" to the calling function, one formally and one by indirection.

Let us write a function to return the square and the cube of a value. We decide that the function returns the square as its value, and "returns" the cube by indirection. We need two parameters; one to pass the value to be squared and cubed to the function, and one pointer type parameter which will be used to indirectly access an appropriate object in the calling function to store the cube of the value. We assume all objects are of type `double`.

The code is shown in Figure 6.13. The prototype for `sqcube()` is defined to have two parameters, a `double` and a pointer to `double`, and it returns a `double` value. The `printf()` prints the value of x; the value of `square` which is the value returned by `sqcube()` (the square of x); and, the value of `cube` (the cube of x) which is indirectly stored by `sqcube()`.

main()

```
        double x              double square        double cube
        ┌──────────┐          ┌──────────┐         ┌──────────┐
        │   3.0    │          │   ? ?    │         │   ? ?    │
        └──────────┘          └──────────┘         └──────────┘
```

sqcube(double [   ]   **double *** [•] **)**

```
                     x                                   pcube
```

                              ┊ **double**
                              ┊
                              ▼
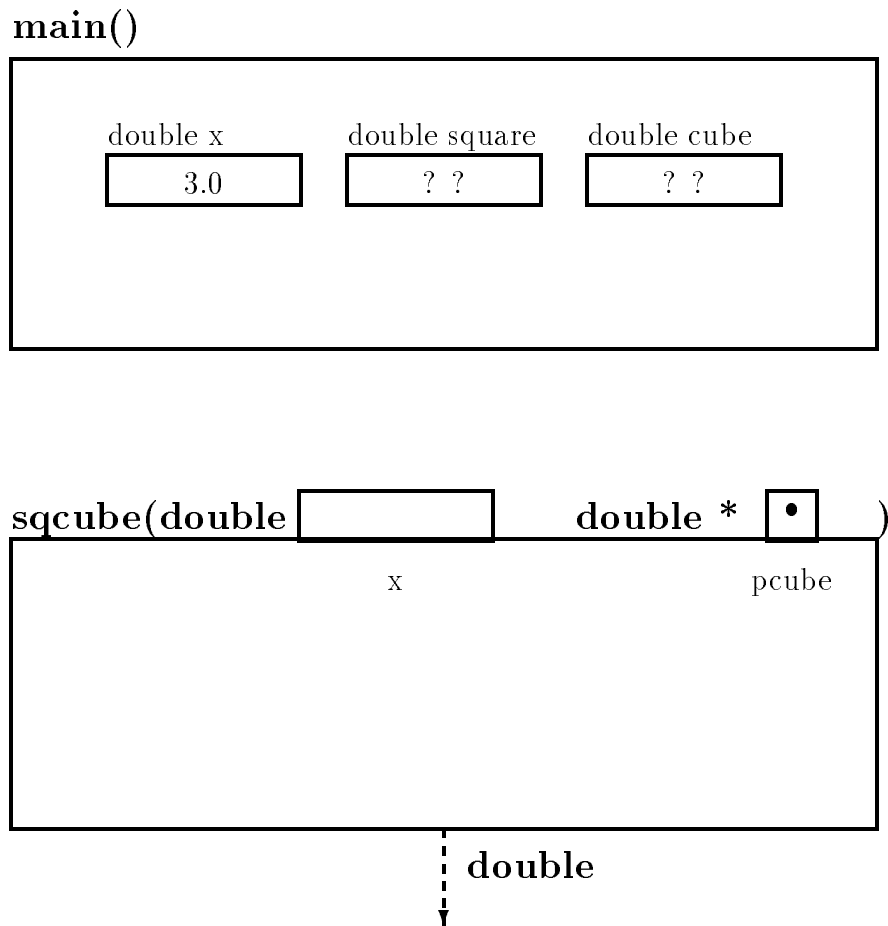
Figure 6.14: Trace for `sqcube` — Step 1

main()

double x   double square   double cube

| 3.0 | | ? ? | | ? ? |

sqcube(double | 3.0 | **double *** | • | **)**

x   pcube

double
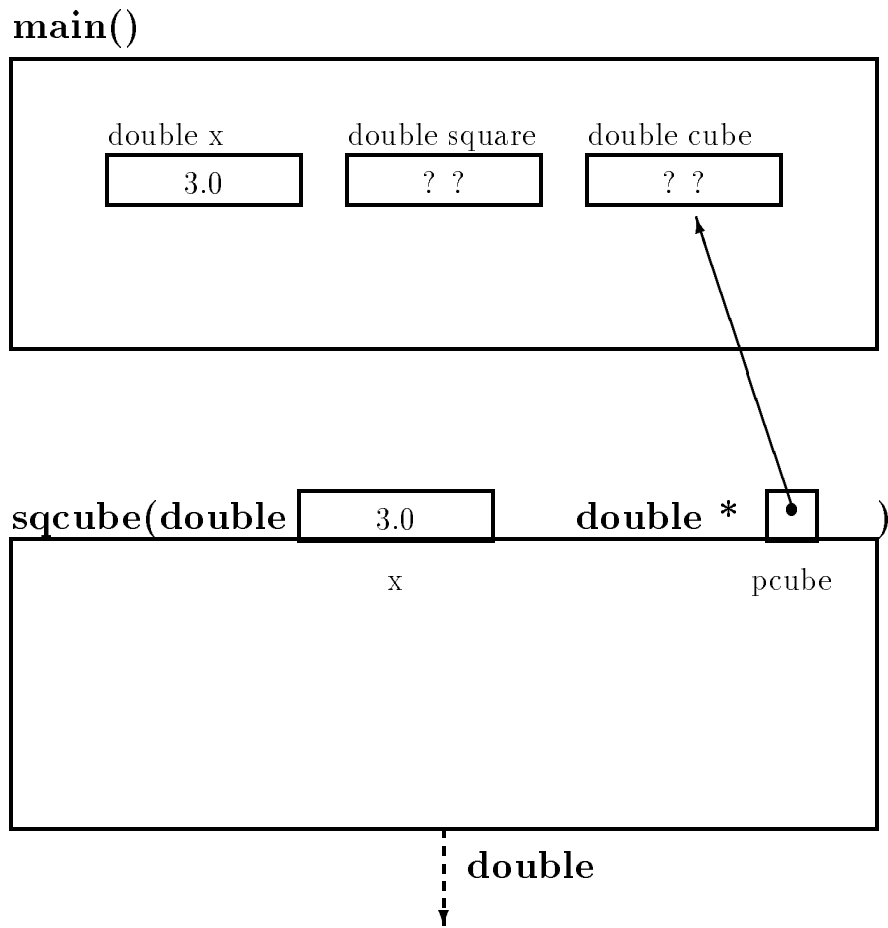
Figure 6.15: Trace for `sqcube` — Step 2

main()

double x   double square   double cube

3.0   ? ?   27.0

sqcube(double | 3.0 | **double *** | • | **)**

x   pcube

double
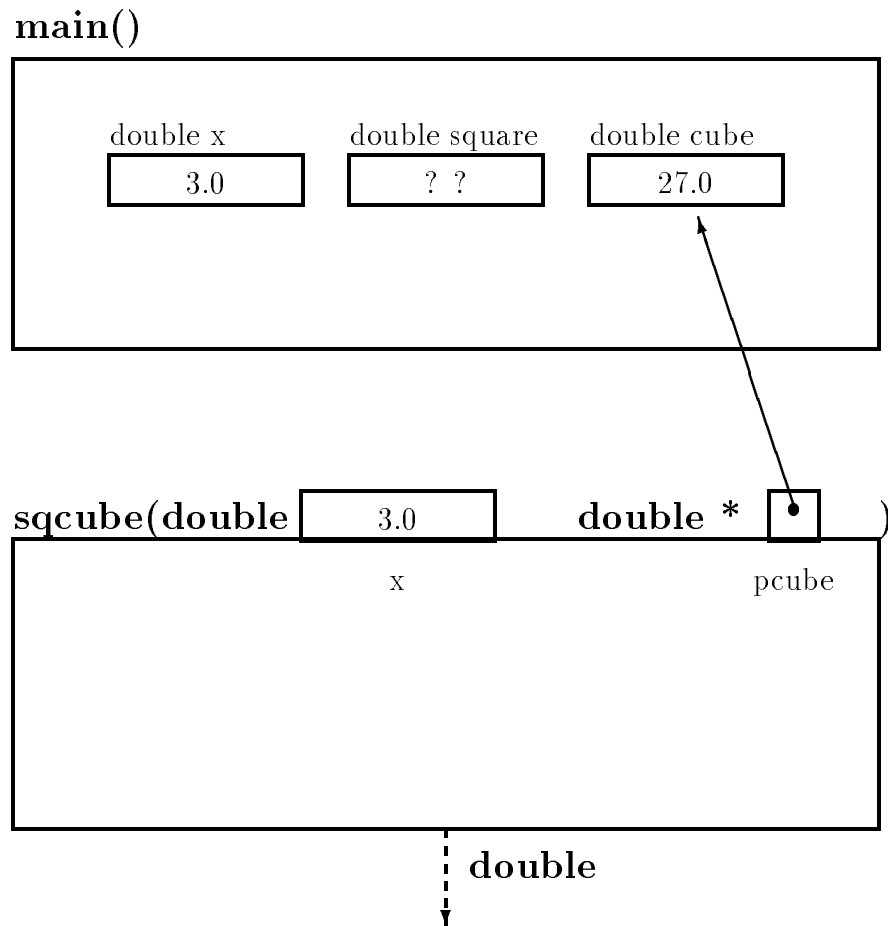
Figure 6.16: Trace for `sqcube` — Step 3

main()

double x
3.0

double square
9.0

double cube
27.0

sqcube(double 3.0 **double *** **)**

x

pcube

double
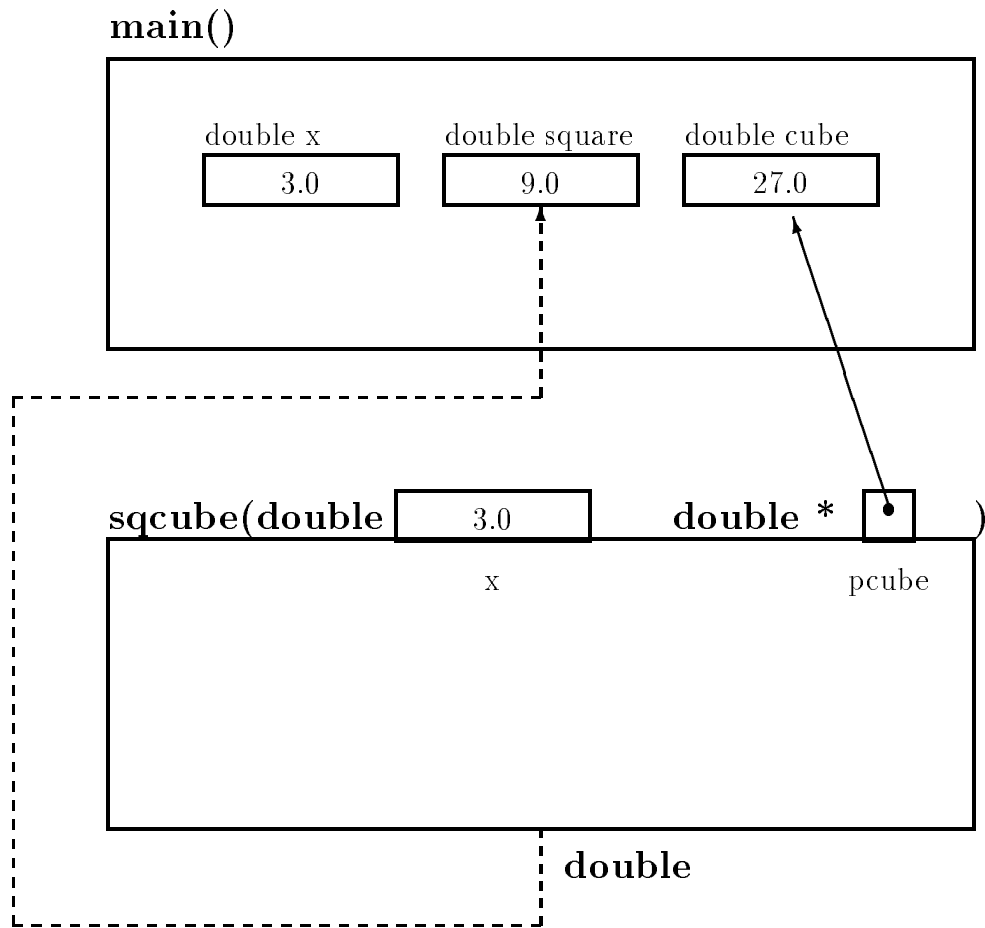
Figure 6.17: Trace for `sqcube` — Step 4

Figures 6.14 — 6.17 show a step-by-step trace of the changes in objects, both in the calling function and in the called function. In the first step (Figure 6.14), the declarations for the function, main() and the template for the function, sqcube() are shown with the initialization of the variable, x, in main(). In the second step (Figure 6.15), the function, sqcube() is called from main() passing the *value* of x (3.0) to the first parameter, (called x in sqcube()), and the *value* of &cube, namely a pointer to cube, as the second argument to the parameter, pcube. In the third step (Figure 6.16), the first statement in sqcube() has been executed, computing the cube of the local variable, x, and storing the value indirectly in the cell pointed to by pcube. Finally, Figure 6.17 shows the situation just as sqcube() is returning, computing the square of x and returning the value which is assigned to the variable, square, by the assignment in main().

While only one value can be returned as the value of a function, we loosely say that this function "returns" two values: the square and the cube of x. The distinction between a formally returned value and an indirectly or loosely "returned" value will be clear from the context.

Sample Session:

```
***Directly and Indirectly Returned Values***
x = 3.000000, square = 9.000000, cube = 27.000000
```

6.2.3   A function to Swap Values

We have already seen how values of two objects can be swapped directly in the code in main(). We now write a function, swap(), that swaps values of two objects defined in main() (or any other function) by accessing them indirectly, i.e. through pointers. The function main() calls the function, swap(), passing pointers to the two variables. The code is shown in Figure 6.18. (We assume integer type objects in main()).

The function, swap(), has two formal parameters, integer pointers, ptr1 and ptr2. A temporary variable is needed in the function body to save the value of one of the objects. The objects are accessed indirectly and swapped. Figures 6.19 — 6.22 show the process of function call, passed values, and steps in the swap.

Sample Session:

```
Original values:  dat1 = 100, dat2 = 200
Swapped values:   dat1 = 200, dat2 = 100
```

6.3   Returning to the Payroll Task with Pointers

We will now modify our pay calculation program so that the driver calls upon other functions to perform *all* subtasks. The driver, main(), represents only the overall logic of the program; the

```
/*   File: swapfnc.c
     Program uses a function to swap values of two objects.
*/
#include <stdio.h>
/* arguments of swap() are integer pointers */
void swap(int * p1, int * p2);


main()
{    int dat1 = 100, dat2 = 200;

     printf("Original values: dat1 = %d, dat2 = %d\n", dat1, dat2);
     swap(&dat1, &dat2);
     printf("Swapped values: dat1 = %d, dat2 = %d\n", dat1, dat2);
}


/*   Function swaps values of objects pointed to by ptr1 and ptr2 */
void swap(int * ptr1, int * ptr2)
{    int temp;

     temp = *ptr1;
     *ptr1 = *ptr2;
     *ptr2 = temp;
}
```

Figure 6.18: Code for a Function, swap()

main()
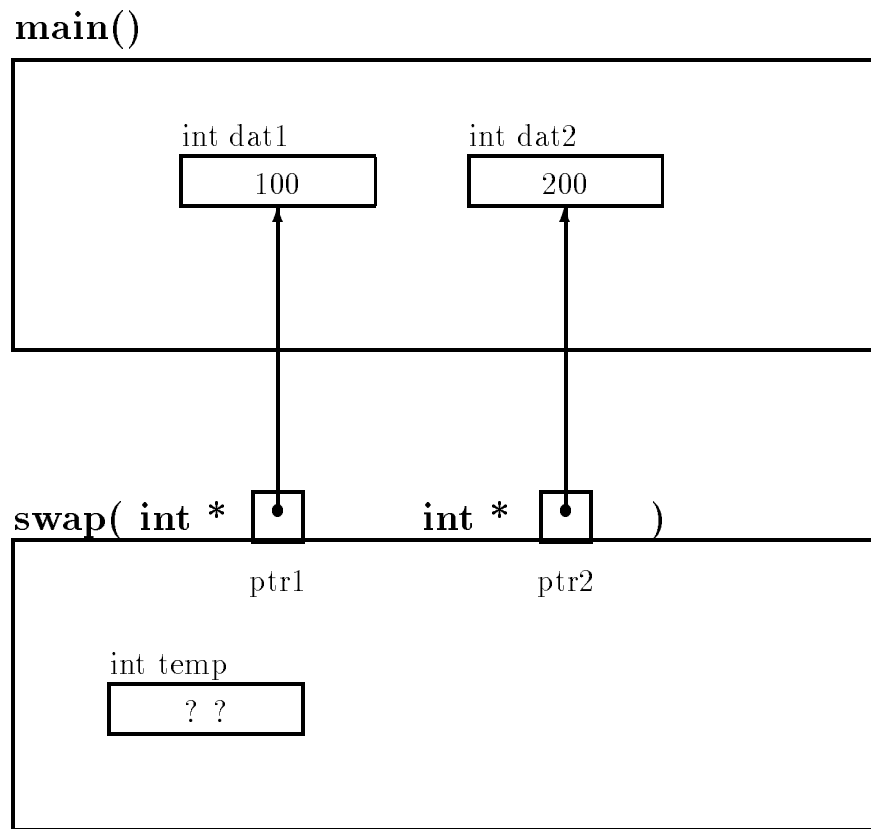
Figure 6.19: Trace for `swap()` — Step 1

main()

int dat1

100

int dat2

200

swap(int * ☐ **int *** ☐ **)**

ptr1   ptr2

int temp

100
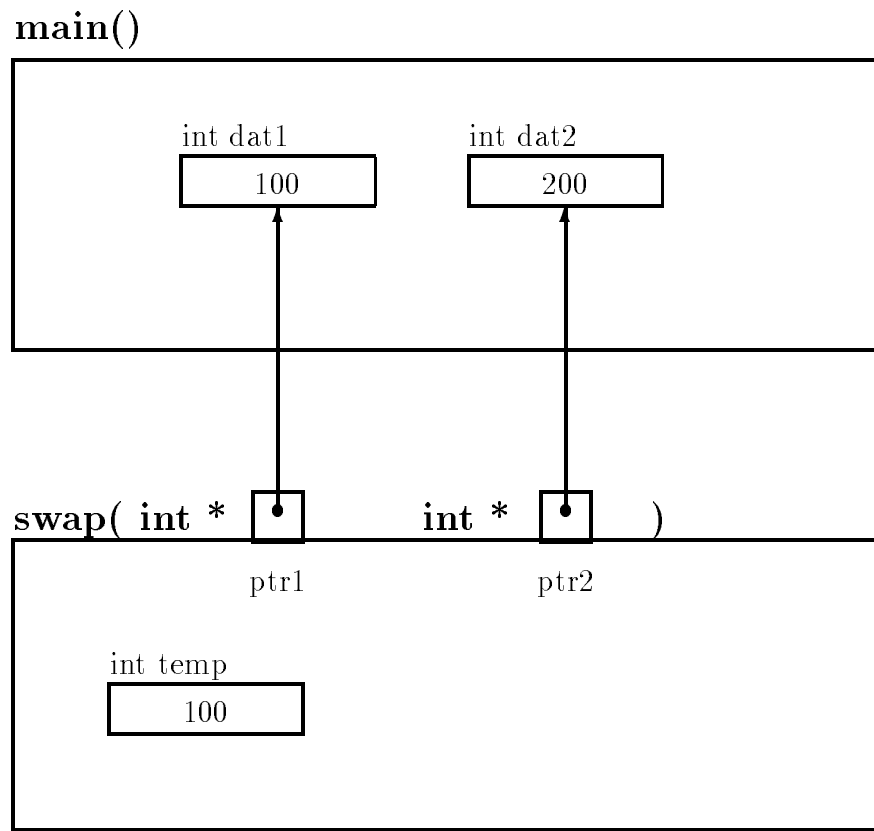
Figure 6.20: Trace for `swap()` — Step 2

main()

int dat1
200

int dat2
200

swap(int * ⬚ **int *** ⬚ **)**

ptr1

ptr2

int temp
100
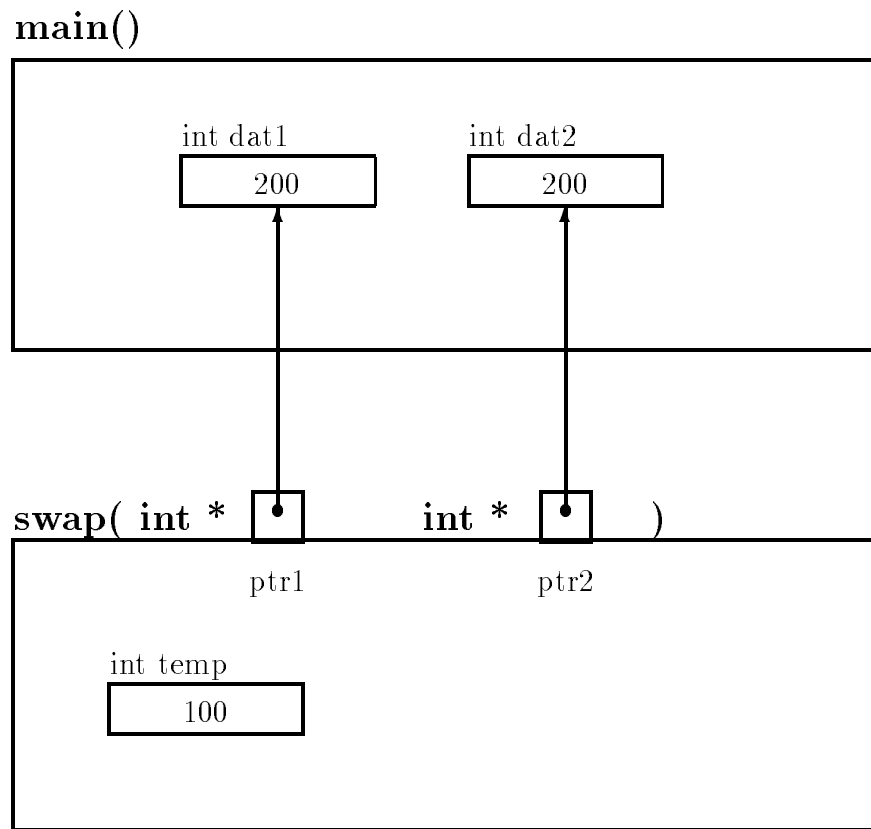
Figure 6.21: Trace for swap() — Step 3

main()

int dat1

200

int dat2

100

swap(int * ▪ **int *** ▪ **)**

ptr1

ptr2

int temp

100
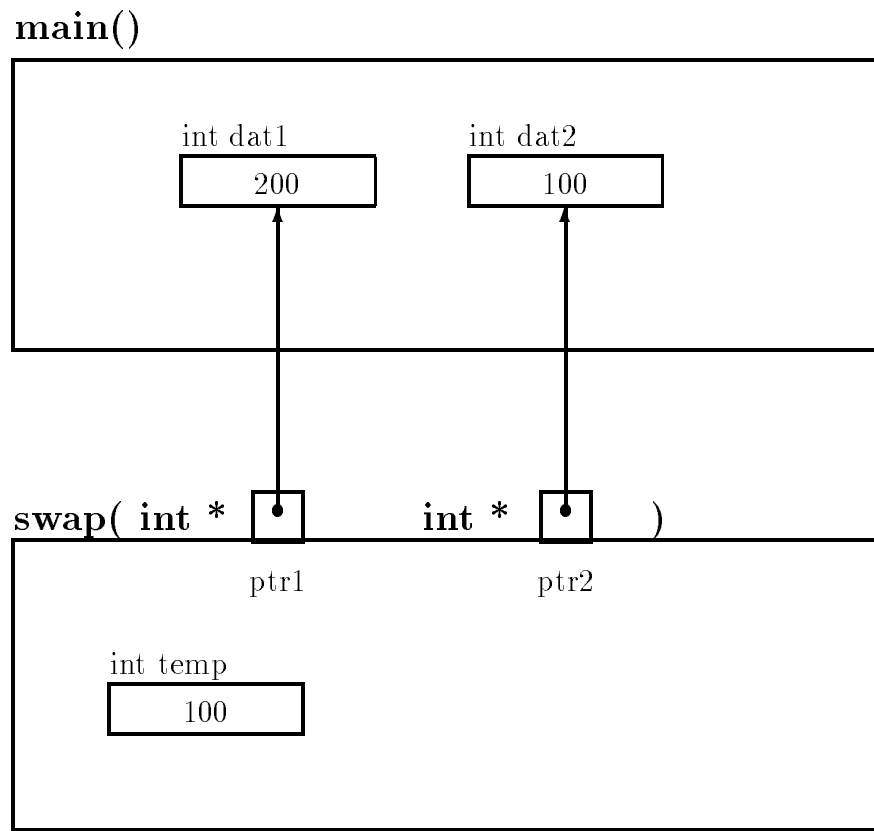
Figure 6.22: Trace for `swap()` — Step 4

details are hidden in the functions that perform the various subtasks. The algorithm for the driver
is:

```
get data
repeat the following while there is more data
    calculate pay
    print data and results
    get data
```

For each step of the algorithm, we will use functions to do the tasks of getting data, printing
data and results, and calculating pay. We have already written functions in Chapters 3 and 4 to
calculate pay and to print data and results, and will repeat them here for easy reference, making
some modifications and improvements. We have postponed until now writing a function to read
data as such a function would require returning more than one value. By using pointers, we now
have the tool at our disposal to implement such a function.

Before we *write* these functions, we should *design* them by describing what the functions do
and specifying the **interface** to these functions; i.e. by indicating the arguments and their types
to be passed to the functions (the information *given* to the functions) and the meaning and type
of the return values (the information returned from the function). Here are our choices:

get_data(): This function reads the id number, hours worked, and rate of pay for one employee
and stores their values indirectly using pointers. Since these values are returned indirectly,
the arguments must be pointers to appropriate objects in the calling function (`main()` in
our case). The function returns True, if it found new data in the input; it returns False
otherwise. Here is the prototype:

```
int get_data(int * pid, float * phrs, float * prate);
```

We use names `pid`, `phrs`, and `prate`, to indicate that they are pointers to cells for the id,
hours and rate, respectively. It is a good habit to distinguish between object names and
pointer names whenever there is a possibility of confusion.

print_data(): This function writes the id number, hours worked, and rate of pay passed to it. It
has no useful information to return so returns a **void** type. Here is the prototype:

```
void print_data(int id, float hrs, float rate, float pay);
```

print_pay(): This function is given values for the regular pay, overtime pay, and total pay and
writes them to the output. It also returns a **void** type.

```
void print_pay(float regular, float overtime, float total);
```

calc_pay(): Given the necessary information (hours and rate), this function calculates and returns
the total pay, and indirectly returns the regular and overtime pay. In addition to the values
of hours worked and rate of pay, pointers to regular pay and overtime pay are passed to the
function. The prototype is:

```
/* File: payutil.h */
#define REG_LIMIT 40
#define OT_FACTOR 1.5
int get_data(int *pid, float *phrs, float *prate);
void print_data(int id, float hrs, float rate);
void print_pay(float regular, float overtime, float total);
float calc_pay(float hours, float rate, float * pregular,
                        float * povertime);
```

Figure 6.23: Header file `payutil.h`

```
float calc_pay(float hours, float rate, float * pregular,
                        float * povertime);
```

Here, `pregular` and `povertime` are pointers to cells for regular and overtime pay objects in the calling function.

All of these functions will be defined in a file, `payutil.c` and their prototypes are included in `payutil.h`. Figure 6.23 shows the header file. We have also included the definitions for symbolic constants `REG_LIMIT` and `OT_FACTOR` in the header file. This header file will be included in all relevant source files.

With the information in this file (and the preceding discussion of the function) we have sufficient information to write the driver for the program *using* the functions prior to writing the actual code for them. Figure 6.24 shows the driver. It also includes the file, `tfdef.h` which defines the macros, `TRUE` and `FALSE`.

The logic of the driver is as follows. After the program title is printed, the first statement calls `get_data()` to get the `id_number`, `hours_worked`, and `rate_of_pay`. As indicated in the prototype, pointers to these objects are passed as arguments so that `get_data()` can indirectly access them and store values. The function, `get_data()`, returns True or False depending on whether there is new data. The True/False value is assigned to the variable, `moredata`. The `while` loop is executed as long as there is more data; i.e. `moredata` is True. The loop body calls on `calc_pay()` to calculate the pay, `print_data()` to print the input data, `print_pay()` to print the results, and `get_data()` again to get more data. Since `calc_pay()` returns the values of overtime and total pay indirectly, `main()` passes pointers to objects which will hold these values.

The overall logic in the driver is easy to read and understand; at this top level of logic, the details of the computations are not important and would only complicate understanding the program. The driver will remain the same no matter how the various functions are defined. The actual details in one or more functions may be changed at a later time without disturbing the driver or the other functions. This program is implemented in functional modules. Such a modular programming style makes program development, debugging and maintenance much easier.

```
/*    File: pay6.c
      Other Files: payutil.c
      Header Files; tfdef.h, payutil.h
    The program gets payroll data, calculates pay, and prints out
    the results for a number of people. Modular functions are used
    to get data, calculate total pay, print data, and print results.
*/
#include <stdio.h>
#include "tfdef.h"
#include "payutil.h"
main()
{
    /* declarations */
    int id_number, moredata;
    float hours_worked, rate_of_pay, regular_pay, overtime_pay, total_pay;


    /* print title */
    printf("***Pay Calculation***\n\n");

    /* get data and initialize loop  variable */
    moredata = get_data(&id_number, &hours_worked,
                          &rate_of_pay);
    /* process while moredata */
    while (moredata) {
        total_pay = calc_pay(hours_worked, rate_of_pay, &regular_pay,
                                  &overtime_pay);
        print_data(id_number, hours_worked, rate_of_pay);
        print_pay(regular_pay, overtime_pay, total_pay);
        moredata = get_data(&id_number, &hours_worked,
                              &rate_of_pay);
    }
}
```

Figure 6.24: Code for the Driver for pay6.c

```
/* File: payutil.c */
#include <stdio.h>
#include "tfdef.h"
#include "payutil.h"
/* Function prints out the input data */
void print_data(int id, float hours, float rate)
{
    printf("\nID Number = %d\n", id);
    printf("Hours Worked = %f, Rate of Pay = %f\n",
              hours, rate);
}


/* Function prints pay data */
void print_pay(float regular, float overtime, float pay)
{
    printf("Regular Pay = %f, Overtime Pay = %f\n",
              regular, overtime);
    printf("Total Pay = %f\n", pay);
}
```

Figure 6.25: Code for print_data() and print_pay()

Of course, we still have to write the various functions used in the above driver. We write each of these functions in turn. Figure 6.25 shows the code for print_data() and print_pay() in the file payutil.c which are simple enough.

The next two functions require indirect access. The function, calc_pay(), must indirectly store the regular and overtime pay so the formal parameters include two pointers: preg (pointing to the cell for regular pay) and pover (pointing to the cell for overtime pay). The function returns the value of the total pay. It is shown in Figure 6.26. Finally, get_data() must indirectly store the values of the id number, hours worked, and rate of pay, and return True if id number is positive, and False otherwise. Figure 6.27 shows the code. The formal parameters pid, phrs, and prate are pointers to objects in the calling function (main() in our case). Recall, when scanf() is called to read data, it requires arguments that are pointers to the objects where the data is to be placed so that it can indirectly store the values. Therefore, when get_data() calls scanf(), it must pass pointers to relevant objects as arguments, i.e. it passes the pointers, pid, phrs, and prate. These pointer variables point to the objects where values are to be stored. We do NOT want to pass &pid, &phrs, &prate as these are the addresses of the pointers, pid, phrs, and prate; they are NOT the addresses cells to hold the data. If the id number stored in *pid is not positive, i.e. (*pid <= 0), get_data() returns FALSE to indicate that there is no more data. If *pid is positive, the rest of the function is executed, in which case the rest of the input data is read. The value, TRUE is returned to indicate that more data is present.

The above functions are in the source file, payutil.c which must be compiled and linked with the source program file, pay6.c. A sample session would be similar to the ones for similar previous

```
/* File: payutil.c - continued */
/* Function calculates and returns total pay */
float calc_pay(float hours, float rate, float * preg, float * pover)
{    float total;

     if (hours > REG_LIMIT) {
          *preg = REG_LIMIT * rate;
          *pover = OT_FACTOR * rate * (hours - REG_LIMIT);
     }
     else {
          *preg = hours * rate;
          *pover = 0;
     }
     total = *preg + *pover;
     return total;
}
```

Figure 6.26: Code for `calc_pay()`

```
/* File: payutil.c - continued */
/* Function reads in the payroll data */
int get_data(int * pid, float * phrs, float * prate)
{
     printf("Type ID Number, zero to quit: ");
     scanf("%d", pid);
     if (*pid <= 0)              /* if ID number is <= 0, */
          return FALSE;          /* return 0 to calling function */
     printf("Hours Worked: "); /* ID number is valid, get data */
     scanf("%f", phrs);
     printf("Hourly Rate: ");
     scanf("%f", prate);
     return TRUE;                /* valid data entered, return 1 */
}
```

Figure 6.27: Code for `get_data()`

programs and is not shown here.

6.4 Common Errors

1. Using an uninitialized pointer. Remember, declaring a pointer variable simply allocates a cell that can hold a pointer — it does not place a value in the cell. So, for example, a code fragment like:

```
{  int * iptr;

   *iptr = 2;
   . . .
}
```

will attempt to place the value, 2, in the cell pointed to by `iptr`; however, `iptr` has not been initialized, so some garbage value will be used as the address of there to place the value. On some systems this may result in an attempt to access an illegal address, and a memory violation. Avoid this error by remembering to initialize all pointer variables before they are used.

2. Instead of using a pointer to an object, a pointer to a pointer is used. Consider a function, `read_int()`. It reads an integer and stores it where its argument points. The correct version is:

```
void read_int(int * pn)
{
      scanf("%d", pn);
}
```

`pn` is a pointer to the object where the integer is to be stored. When passing the argument to `scanf()`, we pass the pointer, `pn`, NOT `&pn`.

3. Confusion between the address of operator and the dereference operator.

```
... calling_func(...)
{    int x;
     called_func(*x);         /* should be &x   */
     ...
}
... called_func(int &px)      /* should be * px */
{
     ...
}
```

A useful mnemonic aid is that the "address of" operator is the "and" symbol, `&` — both start with letter, *a*.

6.5   Summary

In this Chapter we have introduced a new data type, a **pointer**. We have seen how we can declare variables of this type using the * and indicating the type of object this variable can point to, for example:

```
int   * iptr;
float * fptr;
char  * cptr;
```

declare three pointer variables, `iptr` which can point to an integer cell, `fptr` which can point to a cell holding a floating point variable, and `cptr` which can point to a character cell.

We have seen how we can assign values to pointer variables using the "address of" operator, **&** as well as from other pointer variables. For example,

```
{   int x;
    int * ip;
    int * iptr;

    iptr = &x;
    ip = iptr;

    .  .  .
}
```

declares an integer variable, `x`, and two integer pointers, `ip` and `iptr`, which can point to integers (we can read this last declaration from right to left, as saying that " `iptr` points to an `int`"). We then assign the *address* of `x` to the pointer variable, `iptr`, and the *pointer* in `iptr` to the variable, `ip`.

We have also shown how pointer variables may be used to *indirectly* access the value in a cell using the **dereference** operator, *:

```
y = *iptr;
```

which assigns the value of the cell pointed to by `iptr` to the variable, `y`. Values may also be stored indirectly using pointer variables:

```
*iptr = y;
```

which assigns the value in the variable, `y`, to the cell pointed to by `iptr`.

We have also seen that we can pass pointers to functions and use them to modify the values of cells in the *calling* function. For example:

```
main()
{  int x, y, z;

    z = set_em( &x, &y};

    . . .
}

int set_em( int *a, int *b)
{
    *a = 1;
    *b = 2;
    return 3;
}
```

Here the function, set_em will set the values 1, 2, and 3 into the variables x, y, and z respectively.
The first two values are assigned indirectly using the pointers passed to the function, and the third
is returned as the value of the function and assigned to z by the assignment statement in main().
This, the function, set_em(), has "effectively" returned three values.

Finally, we have used this new indirect access mechanism to write several programs, including
an update to our payroll program. As we will see in succeeding chapters, pointers are very useful
in developing complex programs. The concept of pointers may be confusing at first, however, a
useful tool for understanding the behavior of a program using pointers is to draw the memory
picture showing which to cells each pointer is pointing.

6.6   Exercises

1. What is the output of the following code?

```
int x, y, z, w;
int * pa, * pb, * pc, * pd;

x = 10; y = 20; z = 30;
pa = &x;
pb = &y;

printf("%d, %d, %d\n", *pa, *pb, *pc);
pc = pb;
printf("%d, %d, %d\n", *pa, *pb, *pc);
pb = pa;
printf("%d, %d, %d\n", *pa, *pb, *pc);
pa = &z;
printf("%d, %d, %d\n", *pa, *pb, *pc);
*pa = *pb;
printf("%d, %d, %d\n", *pa, *pb, *pc);
```

What is the output for each of the following programs:

2.
```
#define SWAP(x, y)  {int temp; temp = x; x = y; y = temp; )
main()
{    int data1 = 10, data2= 20;
     SWAP(data1, data2);
     printf("Data1 = %d, data2 = %d\n", data1, data2);
}
```

3.
```
#define SWAP(x, y)  {int *temp; temp = x; x = y; y = temp; )
main()
{    int data1 = 10, data2= 20;
     int *p1, *p2;
     p1 = &data1; p2 = &data2;
     SWAP(p1, p2);
     printf("*p1 = %d, *p2 = %d\n", *p1, *p2);
}
```

Correct the code in the following problems:

4.
```
main()
{    int x, *p;

     x = 13;
     ind_square(*p);
}
```

```
            ind_square(int *p)
            {
                  *p = *p * *p;
            }

5.          main()
            {     int x, *p;

                  x = 13; p = &x;
                  ind_square(&p);
            }

            ind_square(int &p)
            {
                  *p = *p * *p;
            }

6.          main()
            {     int x, *p;

                  x = 13;
                  ind_square(x);
            }

            ind_square(int *p)
            {
                  *p = *p * *p;
            }

7.          main()
            {     int x, *p;

                  x = 13;
                  ind_square(p);
            }

            ind_square(int *p)
            {
                  *p = *p * *p;
            }
```

6.7   Problems

1. Write a program that initializes integer type variables, `data1` and `data2` to the values 122 and 312. Declare pointers, `ptr1` and `ptr2`; initialize `ptr1` to point to `data1` and `ptr2` to point to `data2`. Swap the values of `data1` and `data2` values using direct access and using indirect access. Next, swap the values of the pointers, `ptr1` and `ptr2` and print the values indirectly accessed by the swapped pointers.

2. Write a function (that returns `void`) which reads and indirectly stores three values in the calling function. The types of the three data items are an integer, a character, and a float.

3. Write a function `maxmin(float x, float * pmax, float * pmin)` where `x` is a new value which is to be compared with the largest and the smallest values pointed to by `pmax` and `pmin`, respectively. The function should indirectly update the largest and the smallest values appropriately. Write a program that reads a sequence of numbers and uses the above function to update the maximum and the minimum until end of file, when the maximum and the minimum should be printed.

4. Repeat Problem 2.10 using functions `get_course_data()`, `calc_gpr()`, and `print_gpr()`.

5. Rewrite Problem 5.1 as a function that finds the roots of a quadratic and returns them indirectly.

6. Rewrite the program to solve simultaneous equations (Problem 5.10). The program should use a function, `solve_eqns()` to solve for the unknowns. The function must indirectly access objects in main() to store the solution values.

7. Write a menu-driven program that uses the function, `solve_eqns()`, of Problem 6. The commands are: get data, display data, solve equations, print solution, verify solution, help, and quit. Use functions to implemnent the code for each command.

8. A rational number is maintained as a ratio of two integers, e.g., 20/23, 35/46, etc. Rational number arithmetic adds, subtracts, multiplies and divides two rational numbers. Write a function to add two rational numbers.

9. Write a function to subtract two rational numbers.

10. Write a function to multiply two rational numbers.

11. Write a function to divide two rational numbers.

12. Write a function to reduce a rational number. A reduced rational number is one in which all common factors in the numerator and the denominator have been cancelled out. For example, 20/30 is reduce to 2/3, 24/18 is reduced to 4/3, and so forth.

13. Use the function, `reduce()`, of Problem 12 to implement the functions in Problems 8 through 11.

14. Rewrite the program of Problem 5.13 to calculate the current and the power in a resistor using a function instead to perform the calculations. One value may be returned as a function value, the other must be indirectly stored in the calling function.

Chapter 7

Arrays

A programmer is concerned with developing and implementing algorithms for a variety of tasks. As tasks become more complex, algorithm development is facilitated by structuring or organizing data in specialized ways. There is no best data structure for all tasks; suitable data structures must be selected for the specific task. Some data structures are provided by programming languages; others must be derived by the programmer from available data types and structures.

So far we have used integer, floating point and character data types as well as pointers to them. These data types are called **base** or **scalar** data types. Such base data types may be used to derive data structures which are organized groupings of instances of these types. The C language provides some widely used **compound** or **derived** data types together with mechanisms which allow the programmer to define variables of these types and access the data stored within them.

The first such type we will discuss is called an **array**. Many tasks require storing and processing a list of data items. For example, we may need to store a list of exam scores and to process it in numerous ways: find the maximum and minimum, average the scores, sort the scores in descending order, search for a specific score, etc. Data items in simple lists are usually of the same scalar type; for example a list of exam scores consists of all integer type items. We naturally think of a list as a data structure that should be referenced as a unit. C provides a derived data type that stores such a list of objects where each object is of the same data type — the array.

In this chapter, we will discuss arrays; how they are declared and data is accessed in an array. We will discuss the relationship between arrays and pointers and how arrays are passed as arguments in function calls. We will present several example programs using arrays, including a revision of our "payroll" task from previous chapters. One important use of arrays is to hold strings of characters. We will introduce strings in this chapter and show how they are stored in C; however, since strings are important in handling non-numeric data, we will discuss string processing at length in Chapter 10.

7.1   A Compound Data Type — *array*

As described above, an array is a compound data type which allows a collection of data of the
same type to be grouped into a single object. As with any data type, to understand how to use an
array, one must know how such a structure can be declared, how data may be stored and accessed
in the structure, and what operations may be performed using this new type.

7.1.1   Declaring Arrays

Let us consider the task of reading and printing a list of exam scores.

   LIST0: Read and store a list of exam scores and then print it.

   Since we are required to store the entire list of scores before printing it, we will use an array
hold the data. Successive elements of the list will be stored in successive elements of the array.
We will use a counter to indicate the next available position in the array. Such a counter is called
an **index** into the array. Here is an algorithm for our task:

```
initialize the index to the beginning of the array
while there are more data items
    read a score and store in array at the current index
    increment index
set another counter, count = index - the number of items in the array
traverse the array: for  each index starting at the beginning to count
    print the array element at index
```

The algorithm reads exam scores and stores them in successive elements of an array. Once the list
is stored in an array, the algorithm traverses the array, i.e. accesses successive elements, and prints
them. A count of items read in is kept and the traversal continues until that count is reached.

   We can implement the above algorithm in a C program as shown in Figure 7.1. Before ex-
plaining this code, here is a sample session generated by executing this program:

```
***List of Exam Scores***

Type scores, EOF to quit
67
75
82
69
^D


***Exam Scores***
```

```
/*  File: scores.c
    This program reads a list of integer exam scores and prints them out.
*/
#include <stdio.h>
#define MAX 100

main()
{   int exam_scores[MAX], index, n, count;

    printf("***List of Exam Scores***\n\n");
    printf("Type scores, EOF to quit\n");

    /* read scores and store them in an array */
    index = 0;
    while ((index < MAX) && (scanf("%d", &n) != EOF))
        exam_scores[index++] = n;
    count = index;

    /* print scores from the array */
    printf("\n***Exam Scores***\n\n");
    for (index = 0; index < count; index++)
        printf("%d\n", exam_scores[index]);
}
```

Figure 7.1: Code for scores.c

int exam_scores[MAX]

index   **subscripted expression**
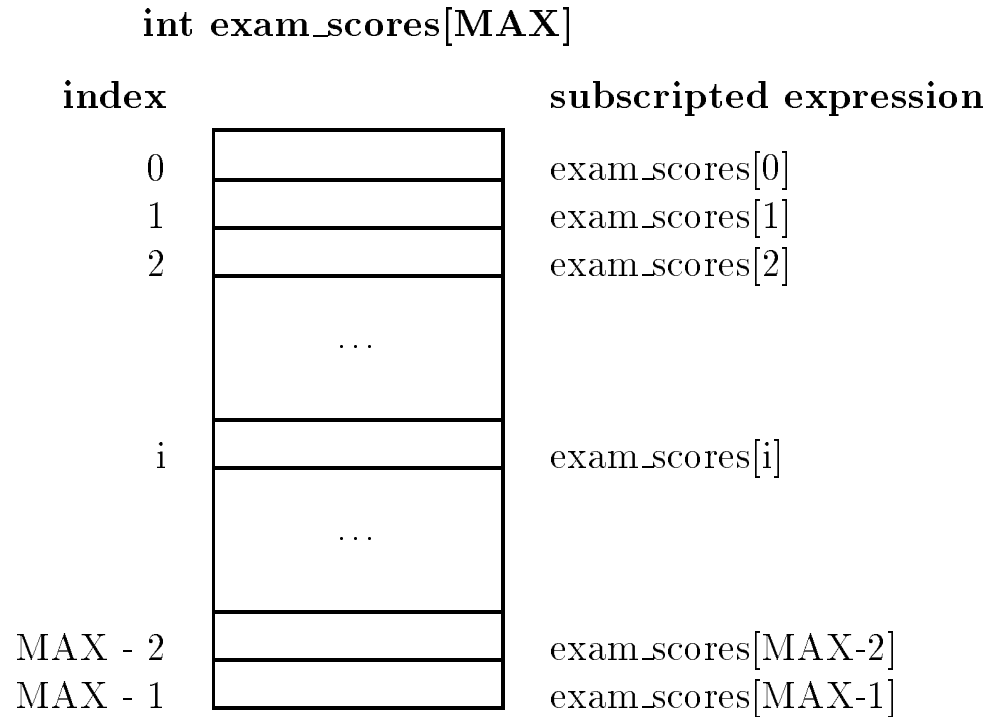
Figure 7.2: An Array of size MAX

```
67
75
82
69
```

Referring to the code in Figure 7.1, the program first declares an array, `exam_scores[MAX]`, of type integer. This declaration allocates a contiguous block of memory for objects of integer type as shown in Figure 7.2. The macro, `MAX`, in square brackets gives the **size** of the array, i.e. the number of elements this compound data structure is to contain. The name of the array, `exam_scores`, refers to the entire collection of `MAX` integer cells. Individual objects in the array may be accessed by specifying the name of the array and the **index**, or element number, of the object; a process called **indexing**. In C, the elements in the array are numbered from 0 to MAX - 1. So, the elements of the array are referred to as `exam_scores[0]`, `exam_scores[1]`, ..., `exam_scores[MAX - 1]`, where the index of each element is placed in square brackets. These index specifiers are sometimes called **subsctipts**, analogous to the mathematical expression $exam\_scores_i$a. These *indexed* or *subscripted* array expressions are the *names* of each object in the array and may be used just like any other variable name.

In the code, the while loop reads a score into the variable, `n`, places it in the array by assigning it to `exam_scores[index]`, and increments `index`. The loop is terminated either when `index` reaches `MAX` (indicating a full array) or when `scanf()` returns `EOF`, indicating the end of the data.

We could have also read each data item directly into `exam_scores[index]` by writing `scanf()` as follows:

```
scanf("%d", &exam_scores[index])
```

We choose to separate reading an item and storing it in the array because the use of the increment operator, `++`, for `index` is clearer if reading and storing of data items are separated.

Once the data items are read and stored in the array, a count of items read is stored in the variable `count`. The list is then printed using a `for` loop. The array is traversed from element 0 to element `count - 1`, printing each element in turn.

From the above example, we have seen how we can declare a variable to be of the compound data type, array, how data can be stored in the elements of the array, and subsequently accessed. More formally, the syntax for an array declaration is:

<type-specifier><identifier>[<size>];

where the <type-specifier> may be any scalar or derived data type; and the <size> must evaluate, at compile time, to an unsigned integer. Such a declaration allocates a contiguous block of memory for objects of the specified type. The data type for each object in the block is specified by the <type-specifier>, and the number of objects in the block is given by —sf <size> as seen in Figure 7.2. As stated above, the index values for all arrays in C must start with 0 and end with the highest index, which is one less than the size of the array. The subscripting expression with the syntax:

<identifier>[<expression>]

is the name of one element object and may be used like any other variable name. The subscript, <expression> must evaluate, at run time, to an integer. Examples include:

```
int a[10];
float b[20];
char s[100];
int  i = 0;

a[3] = 13;
a[5] = 8 * a[3];
b[6] = 10.0;
printf("The value of b[6] is %f\n", b[6]);
scanf("%c", &s[7]);
c[i] = c[i+1];
```

Through the remainder of this chapter, we will use the following symbolic constants for many of our examples:

```
/* File: araydef.h */
#define MAX 20
#define SIZE 100
```

In programming with arrays, we frequently need to initialize the elements. Here is a loop that traverses an array and initializes the array elements to zero:

```
int i, ex[MAX];

for (i = 0; i < MAX; i++)
    ex[i] = 0;
```

The loop assigns zero to `ex[i]` until `i` becomes `MAX`, at which point it terminates and the array elements are all initialized to zero. One precaution to programmers using arrays is that C does not check if the index used as a subscript is within the size of the declared array, leaving such checks as the programmer's responsibility. Failure to do so can, and probably will result in catastrophe.

7.1.2   Character Strings as Arrays

Our next task is to store and print non-numeric text data, i.e. a sequence of characters which are called **strings**. A string is an list (or string) of characters stored contiguously with a marker to indicate the end of the string. Let us consider the task:

STRING0: Read and store a string of characters and print it out.

Since the characters of a string are stored contiguously, we can easily implement a string by using an array of characters if we keep track of the number of elements stored in the array. However, common operations on strings include breaking them up into parts (called **substrings**), joining them together to create new strings, replacing parts of them with other strings, etc. There must be some way of detecting the size of a current valid string stored in an array of characters.

In C, a string of characters is stored in successive elements of a character array and terminated by the NULL character. For example, the string `"Hello"` is stored in a character array, `msg[]`, as follows:

```
char msg[SIZE];

msg[0] = 'H';
msg[1] = 'e';
msg[2] = 'l';
msg[3] = 'l';
msg[4] = 'o';
msg[5] = '\0';
```

```
       0    1    2    3    4    5                          index
```
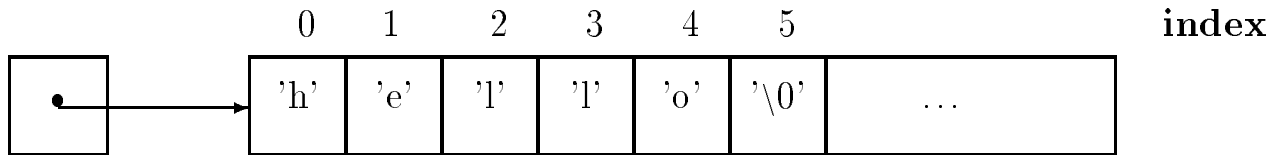

Figure 7.3: A String Stored in Memory

The NULL character is written using the escape sequence '\0'. The ASCII value of NULL is 0, and NULL is defined as a macro to be 0 in stdio.h; so programs can use the symbol, NULL, in expressions if the header file is included. The remaining elements in the array after the NULL may have any garbage values. When the string is retrieved, it will be retrieved starting at index 0 and succeeding characters are obtained by incrementing the index until the first NULL character is reached signaling the end of the string. Figure 7.3 shows a string as it is stored in memory.

Given this implementation of strings in C, the algorithm to implement our task is now easily written. We will assume that a string input is a sequence of characters terminated by a newline character. (The newline character is not part of the string). Here is the algorithm:

```
initialize index to zero
while not a newline character
      read and store a character in the array at the next index
      increment the index value
terminate the string of characters in the array with a NULL char.
initialize index to zero
traverse the array until a NULL character is reached
      print the array character at index
      increment the index value
```

The program implementation has:

- a loop to read string characters until a newline is reached;

- a statement to terminate the string with a NULL;

- and a loop to print out the string.

The code is shown in Figure 7.4 and a sample session form the program is shown below.

Sample Session:

```
***Character Strings***

Type characters terminated by a RETURN or ENTER
```

```
/*  File: string.c
    This program reads characters until a newline, stores them in an
    array, and terminates the string with a NULL character. It then prints
    out the string.
*/

#include <stdio.h>
#include "araydef.h"

main()
{   char msg[SIZE], ch;
    int i = 0;

    printf("***Character Strings***\n\n");
    printf("Type characters terminated by a RETURN or ENTER\n");

    while ((ch = getchar()) != '\n')
        msg[i++] = ch;

    msg[i] = '\0';

    i = 0;
    while (msg[i] != '\0')
        putchar(msg[i++]);
    printf("\n");
}
```

Figure 7.4: Code for string.c

```
Hello
Hello
```

The first `while` loop reads a character into `ch` and checks if it is a newline, which discarded and the loop terminated. Otherwise, the character is stored in `msg[i]` and the array index, `i`, incremented. When the loop terminates, a `NULL` character is appended to the string of characters. In this program, we have assumed that the size of `msg[]` is large enough to store the string. Since a line on a terminal is 80 characters wide and since we have defined SIZE to be 100, this seems a safe assumption.

The next `while` loop in the program traverses the string and prints each character until a `NULL` character is reached. Note, we do not need to keep a count of the number of characters stored in the array in this program since the first `NULL` character encountered indicates the end of the string. In our program, when the first `NULL` is reached we terminate the string output with a newline.

The assignment expression in the above program:

```
msg[i] = '\0';
```

can also be written as:

```
msg[i] = NULL;
```

or:

```
msg[i] = 0;
```

In the first case, the character whose ASCII value is 0 is assigned to `msg[i]`; where in the other cases, a zero value is assigned to `msg[i]`. The above assignment expressions are identical. The first expression makes it clear that a null character is assigned to `msg[i]`, but the second uses a symbolic constant which is easier to read and understand.

To accommodate the terminating `NULL` character, the size of an array that houses a string must be at least one greater than the expected maximum size of string. Since different strings may be stored in an array at different times, the first `NULL` character in the array delimits a valid strin. The importance of the `NULL` character to signal the end of a valid string is obvious. If there were no `NULL` character inserted after the valid string, the loop traversal would continue to print values interpreted as characters, possibly beyond the array boundary until it fortuitously found a `NULL` (0) character.

The second `while` loop may also be written:

```
while (msg[i] != NULL)
    putchar(msg[i++]);
```

and the `while` condition further simplified as:

```
while (msg[i])
    putchar(msg[i++]);
```

If `msg[i]` is any character with a non-zero ASCII value, the `while` expression evaluates to True. If `msg[i]` is the `NULL` character, its value is zero and thus False. The last form of the `while` condition is the more common usage. While we have used the increment operator in the putchar() argument, it may also be used separately for clarity:

```
while (msg[i]) {
    putchar(msg[i]);
    i++;
}
```

It is possible for a string to be empty; that is, a string may have no characters in it. An empty string is a character array with the `NULL` character in the zeroth index position, `msg[0]`.

7.2   Passing Arrays to Functions

We have now seen two examples of the use of arrays — to hold numeric data such as test scores, and to hold character strings. We have also seen two methods for determining how many cells of an array hold useful information — storing a count in a separate variable, and marking the end of the data with a special character. In both cases, the details of array processing can easily obscure the actual logic of a program — processing a set of scores or a character string. It is often best to treat an array as an *abstract data type* with a set of allowed operations on the array which are performed by functional modules. Let us return to our exam score example to read and store scores in an array and then print them, except that we now wish to use functions to read and print the array.

LIST1: Read an array and print a list of scores using functional modules.

The algorithm is very similar to our previous task, except that the details of reading and printing the array is hidden by functions. The function, `read_intaray()`, reads scores and stores them, returning the number of scores read. The function, `print_intaray()`, prints the contents of the array. The refined algorithm for `main()` can be written as:

```
print title, etc.
n = read_intaray(exam_scores, MAX);
print_intaray(exam_scores, n);
```

Notice we have passed an array, `exam_scores`, and a constant, `MAX` (specifying the maximum size of the proposed list), to `read_intarray()` and expect it to return the number of scores placed

in the array. Similarly, when we print the array using `print_intarray`, we give it the array to be printed and a count of elements it contains. We saw in Chapter 6 that in order for a *called* function to access objects in the *calling* function (such as to store elements in an array) we must use *indirect access*, i.e. pointers. So, `read_intaray()` must indirectly access the array, `exam_scores`, in `main()`. One unique feature of C is that array access is *always* indirect; thus making it particularly easy for a called function to indirectly access elements of an array and store or retrieve values. As we will see in later sections, array access by index value is interpreted as an indirect access, so we may simply use array indexing as indirect access.

We are now ready to implement the algorithm for `main()` using functions to read data into the array and to print the array. The code is shown in Figure 7.5. The function calls in `main()` pass the name of the array, `exam_scores`, as an argument because the name of an array in an expression evaluates to a pointer to the array. In other words, the expression, `exam_scores`, is a *pointer* to (the first element of) the array, `exam_scores[]`. Its type is, therefore, `int *`, and a called function uses this pointer (passed as an argument) to indirectly access the elements of the array. As seen in the Figure, for both functions, the headers and the prototypes show the first formal parameter as an integer array without specifying the size. In C, this syntax is interpreted as a pointer variable; so `scores` is declared aa an `int *` variable. We will soon discuss how arrays are accessed in C; for now, we will assume that these pointers may be used to indirectly access the arrays.

The second formal parameter in both functions is `lim` which specifies the maximum number of items. For `read_intaray()`, this may be considered the maximum number of scores that can be read so that it does not read more items than the size of the array allows (`MAX`). The function returns the *actual* number of items read which is saved in the variable, `n`, in `main()`. For the function, `print_intaray()`, `lim` represents the fact that it must not print more than `n` items. Again, since arrays in C are accessed indirectly, these functions are able to access the array which is defined and allocated in `main()`. A sample session for this implementation of the task would be identical to the one shown earlier.

Similarly, we can modify the program, `string.c`, to use functions to read and print strings. The task and the algorithm are the same as defined for STRING0 in the last section, except that the program is terminated when an empty string is read. The code is shown in Figure 7.6. The driver calls `read_str()` and `print_str()` repeatedly until an empty string is read (detected when `s[0]` is zero, i.e. NULL). The argument passed to `read_str()` and `print_str()` is `str`, a pointer to (the first element of) a character array, i.e. a `char *`. The function, `read_str()`, reads characters until a newline is read and indirectly stores the characters into the string, `s`. The function, `print_str()`, prints characters from the string, `s` until NULL is reached and terminates the output with a newline. Notice we have declared the formal parameter, `s` as a `char *`, rather than as an array: `char s[]`. As we will see in the next section, C treats the two declarations exactly the same.

```
/*  File: scores2.c
    This program uses functions to read scores into an array and to print
    the scores.
*/
#include <stdio.h>
#define MAX 10

int read_intaray(int scores[], int lim);
print_intaray(int scores[], int lim);
main()
{   int n, exam_scores[MAX];

    printf("***List of Exam Scores***\n\n");
    n = read_intaray(exam_scores, MAX);
    print_intaray(exam_scores, n);
}


/* Function reads scores in an array. */
int read_intaray(int scores[], int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");

    while ((count < lim) && (scanf("%d", &n) != EOF)) {
        scores[count] = n;
        count++;
    }
    return count;
}


/* Function prints lim elements in the array scores. */
void print_intaray(int scores[], int lim)
{   int i;

    printf("\n***Exam Scores***\n\n");
    for (i = 0; i < lim; i++)
        printf("%d\n", scores[i]);
}
```

Figure 7.5: Code fore `scores.c`

```
/* File: string2.c
   This program reads and writes strings until an empty string is
   read. It uses functions to read and print strings to standard
   files.
*/
#include <stdio.h>
#define SIZE 100
void print_str(char s[]);
void read_str(char s[]);

main()
{  char str[SIZE];

   do {
     read_str(str);
     print_str(str);
   } while (str[0]);
}

/* Function reads a string from standard input until a newline is
   read. A NULL is appended.
*/
void read_str(char *s)
{  int i;
   char c;

   for (i = 0; (c = getchar()) != '\n'; i++)
     s[i] = c;
   s[i] = NULL;
}

/* Function prints a string to standard output and terminates with a
   newline.
*/
void print_str(char *s)
{  int i;

   for (i = 0; s[i]; i++)
     putchar(s[i]);
   putchar('\n');
}
```

Figure 7.6: Code for string2.c

7.3   Arrays, Pointers, Pointer Arithmetic

Let us now examine how arrays are actually accessed in C. As we have seen, an array is a sequence of objects, each of the same data type. The starting address of this array of objects, i.e. the address of the first object in the array is called the **base address** of the array. The address of each successive element of the array is offset from the base by the size of the array type, e.g. for each successive element of an integer array, the address is offset by the size of an integer type object. As we mentioned in the previous section, in C, the name of an array used by itself in an expression evaluates to the base address of the array. That is, this value is a *pointer* type and points to the first object of the array. The name of the array is said to point to the array. Figure 7.7 shows an array, `X[]` with `X` pointing to (the first object of) the array. If the array is an integer array, (float array, character array, etc.) then the type of `X` is `int *` (`float *`, `char *`, etc.). Thus, the declaration of an array causes the compiler to allocate the specified number of contiguous cells of the indicated type, as well as to allocate an appropriate pointer cell, initialized to point to the first cell of the array. This pointer cell is given the name of the array. Since `X` points to `X[0]`, the following are equivalent:

```
    X <---> &X[0]
```

Thus, the dereferenced pointer, `*X`, accesses the object, `X[0]`, i.e. the following are equivalent:

```
    *X <---> X[0]
```

As we have seen, pointer variables point to objects of a specific type. We might suspect that they can be increased or decreased to point to contiguous successive or preceding objects of the same type. In C, adding one to a pointer makes the resulting pointer point to the next object of the same type. (The value of the new pointer equals the original value of the pointer increased by the size of the object pointed to). For the array above, `X + 1` points to `X[1]`; the increase in the pointer value is made by the appropriate size of the type involved. For example, if `X` is an integer array and an integer requires 4 bytes, then the value of `X + 1` will be greater than that of `X` by 4. Adding k to a pointer results in a pointer to a successive object offset by k objects from the original. Thus, `X + 0` points to the start of the array (the first element, `X[0]`), `X + 1` points to the next element, `X[1]`, and `X + k` points to `X[k]` as can be seen in Figure 7.7. Similarly, `&X[k]` is the same as `X + k`, and `X[k]` is the same as `*(X + k)`. Table 7.1 summarizes pointer arithmetic and indirect access of elements of an array. Pointer arithmetic may also involve subtraction; the resulting pointer points to a previous object offset appropriately. Thus, for example, `&X[3] - 1` points to `X[2]`, `&X[5] - 3` points to `X[2]`, and so on.

In C array access is always made through pointers and indirection operators. Whenever an expression such as `X[k]` appears in a program, the compiler interprets it to mean `*(X + k)`. In other words, objects of an array are always accessed indirectly. As we have seen previously, this makes it particularly easy for a called function to indirectly access elements of an array allocated in the calling function to store or retrieve values. Let us see how function calls handle array access using the program, `scores2.c` of the last section. The relevant function calls in `main()` and the corresponding function headers are shown below for easy reference:

X[n]

index   subscripted expression

X or X + 0 ──────────→   X[0]
X + 1 ──────────→   X[1]
X + 2 ──────────→   X[2]

...

X + i ──────────→   X[i]

...

X + n - 2 ──────────→   X[n-2]
X + n - 1 ──────────→   X[n-1]
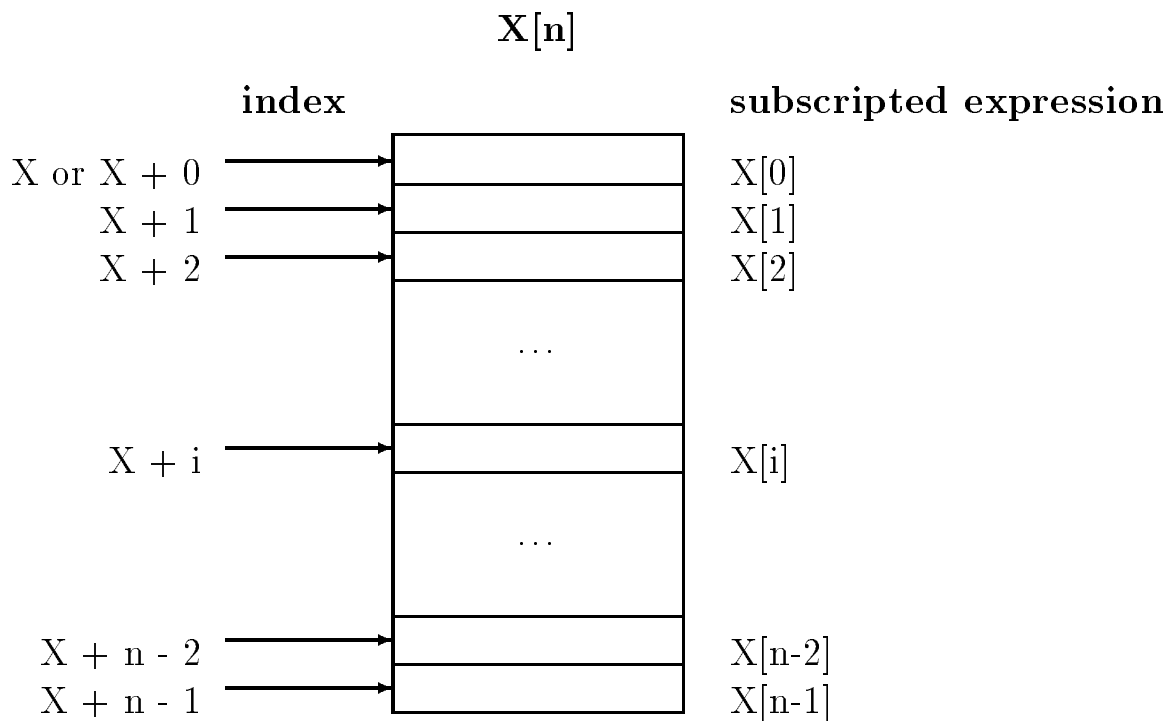
Figure 7.7: Pointer Arithmetic

Pointer Arithmetic	Address of Operator	Array Subscripting	Indirect Reference
X + 0	&X[0]	X[0]	*(X + 0)
X + 1	&X[1]	X[1]	*(X + 1)
X + 2	&X[2]	X[2]	*(X + 2)
X + 3	&X[3]	X[3]	*(X + 3)
...	...	...	...
X + k	&X[k]	X[k]	*(X + k)

Table 7.1: Pointer Arithmetic and Indirect Access

```
main()
{   int exam_scores[MAX];
    ...
    n = read_intaray(exam_scores, MAX);
    print_intaray(exam_scores, n);
}
int read_intaray(int scores[], int lim)
{
    ...
}
void print_intaray(int scores[], int lim)
{
    ...
}
```

When a formal parameter is declared in a function header as an array, it is interpreted as a pointer variable, NOT an array. Even if a size were specified in the formal parameter declaration, only a pointer cell is allocated for the variable, not the entire array. The type of the pointer variable is the specified type. In our example, the formal parameter, `scores`, is an integer pointer. It is initialized to the pointer value passed as an argument in the function call. The value passed in `main()` is `exam_scores`, a pointer to the first element of the array, `exam_scores[]`. Figure 7.8 illustrates the connection between the calling function, `main()`, and the called function, `read_intaray()`. In this case, the formal parameter, `scores`, is initialized to point to the value of `exam_scores` which is a pointer to (the first element of) the array `exam_scores[]`. The Figure also shows that `lim` is initialized to 10.

Within the function, `read_scores()`, it is now possible to access all the elements of the array, `exam_scores[]`, indirectly. Since the variable, `scores`, in `read_intaray()` points to the first element of the array, `exam_scores[]`, `*scores` accesses the first element of the array, i.e. `exam_scores[0]`. In addition, `scores + 1` points to the next element of the array, so `*(scores + 1)` accesses the next element, i.e. `exam_scores[1]`. In general, `*(scores + count)` accesses the element `exam_scores[count]`. To access elements of the array, we can either write `*(scores + count)` or we can write `scores[count]`, because dereferenced array pointers and indexed array elements are identical ways of writing expressions for array access.

The functions, `read_intaray()` and `print_intaray()` can be used to read objects into *any* integer array and to print element values of *any* integer array, respectively. The calling function must simply pass, as arguments, an appropriate array pointer and maximum number of elements.

These functions may also be written explicitly in terms of indirect access, for example:

```
/* Function reads scores in an array. */
int read_intaray2(int * scores, int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");
```
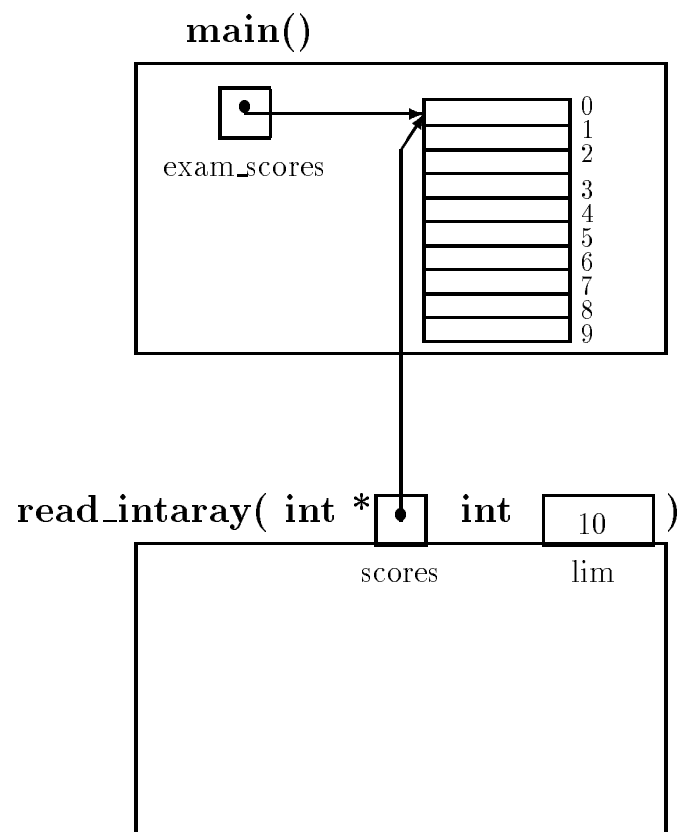
Figure 7.8: Array Pointers as Function Parameters

```
    while ((count < lim) && (scanf("%d", &n) != EOF)) {
        *(scores + count) = n;
        count++;
    }
    return count;
}
```

Alternatively, since `scores` is a pointer variable, we can increment its value each time so that it points to the next object of integer type in the array, such as:

```
/* Function reads scores in an array. */
int read_intaray3(int * scores, int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");
    while ((count < lim) && (scanf("%d", &n) != EOF)) {
        *scores = n;
        count++;
        scores++;
    }
    return count;
}
```

The first time the loop is executed, `*scores` accesses the element of the array at index 0. The *local* pointer cell, `scores`, is then incremented to point to the next element of the array, at index 1. The second time the loop is executed, *scores accesses the array element at index 1. The process continues until the loop terminates.

It is also possible to mix dereferenced pointers and array indexing:

```
/* Function reads scores in an array. */
int read_intaray4(int scores[], int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");
    while ((count < lim) && (scanf("%d", &n) != EOF)) {
        *(scores + count) = n;
        count++;
    }
    return count;
}
```

or:

```
/* Function reads scores in an array. */
```

```
int read_intaray5(int * scores, int lim)
{   int n, count = 0;

    printf("Type scores, EOF to quit\n");
    while ((count < lim) && (scanf("%d", &n) != EOF)) {
        scores[count] = n;
        count++;
    }
    return count;
}
```

We can also consider *parts* of an array, called a **sub-array**. A pointer to a sub-array is also an array pointer; it simply specifies the *base* of the sub-array. In fact, as far as C is concerned, there is no difference between an entire array and any of its sub-arrays. For example, a function call can be made to print a sub-array by specifying the starting pointer of the sub-array and its size. Suppose we wish to print the sub-array starting at `exam_scores[3]` containing five elements; the expression, `&exam_scores[3]` is a pointer to an array starting at `exam_scores[3]`. The function call is:

```
    print_intaray(&exam_scores[3], 5);
```

Alternately, since `exam_scores + 3` points to `exam_scores[3]`, the function call can be:

```
    print_intaray(exam_scores + 3, 5);
```

The passed parameters are shown visually in Figure 7.9. If either of the above function calls is used in the program, `scores2.c`, the values of `exam_scores[3]`, `exam_scores[4]`, ..., and `exam_scores[7]` will be printed.

7.3.1 Pointers: Increment and Decrement

We have just seen that an array name, e.g. `aa`, is a pointer to the array and that `aa + i` points to `aa[i]`. We can illustrate this point in the program below, where the values of pointers themselves are printed. A pointer value is a byte address and is printed as an unsigned integer (using conversion specification for unsigned integer, `%u`). The program shows the relationships between array elements, pointers, and pointer arithmetic.

```
/* File: arayptr.c
   This program shows the relation between arrays and pointers.
*/
#include <stdio.h>
#define N 5
```
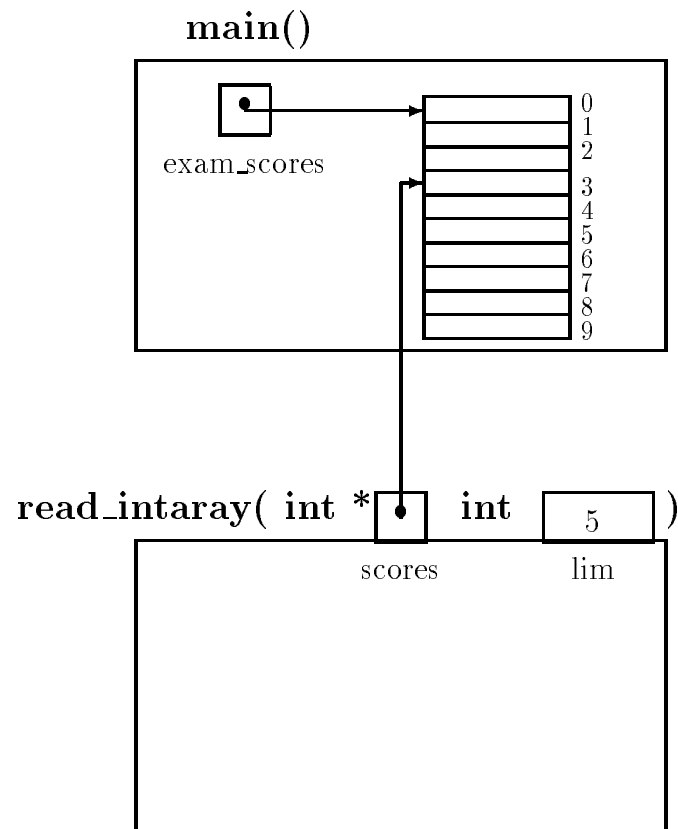
main()

Figure 7.9: Pointer to a Sub-array

```
main()
{   int i, j, aa[N];

    printf("***Pointers, Arrays, and Pointer Arithmetic***\n\n");

    for (i = 0; i < N; i++) {
        aa[i] = i * i;
        printf("aa + %d = %u; &aa[%d] = %u\n", i, aa + i, i, &aa[i]);
        printf("*(aa + %d) = %d; aa[%d] = %d\n", i, *(aa + i), i, aa[i]);
    }
}
```

In the loop, we first assign a value to each `aa[i]`. We then print values to show that pointers, `aa + i` and `&aa[i]` are the same, i.e. that `aa + i` points to `aa[i]`. Next, we print the array element values to show that `*(aa + i)` is the same as `aa[i]`. A sample output for the program is shown below:

```
***Pointers, Arrays, and Pointer Arithmetic***

aa + 0 = 65480; &aa[0] = 65480
*(aa + 0) = 0; aa[0] = 0
aa + 1 = 65482; &aa[1] = 65482
*(aa + 1) = 1; aa[1] = 1
aa + 2 = 65484; &aa[2] = 65484
*(aa + 2) = 4; aa[2] = 4
aa + 3 = 65486; &aa[3] = 65486
*(aa + 3) = 9; aa[3] = 9
aa + 4 = 65488; &aa[4] = 65488
*(aa + 4) = 16; aa[4] = 16
```

(In the host implementation where the above program was executed, two bytes are required for integers; therefore, successive array element addresses are two bytes apart).

The next example shows that pointers may be incremented and decremented. In either case, if the original pointer points to an object of a specific type, the new pointer points to the next or previous object of the same type, i.e. pointers are incremented or decremented in steps of the object size that the pointer points to. Thus, it is possible to traverse an array starting from a pointer to any element in the array. Consider the code:

```
/*  File: arayptr2.c
    Pointers and pointer arithmetic.
*/
#include <stdio.h>
#define N 5
```

```
main()
{   float faray[N], *fptr;
    int *iptr, iaray[N], i;

    /* initialize */
    for (i = 0; i < N; i++) {
        faray[i] = 0.3;
        iaray[i] = 1;
    }

    /* initialize fptr to point to element faray[3] */
    fptr = &faray[3];
    *fptr = 1.;              /* faray[3] = 1. */
    *(fptr - 1) = .9;        /* faray[2] = .9 */
    *(fptr + 1) = 1.1;       /* faray[4] = 1.1 */

    /* initialize iptr in the same way */
    iptr = &iaray[3];
    *iptr = 0;
    *(iptr - 1) = -1;
    *(iptr + 1) = 2;

    for (i = 0; i < N; i++) {
        printf("faray[%d] = %f   ", i, *(faray + 1));
        printf("iaray[%d] = %d\n", i, iaray[i]);
    }
}
```

The program is straightforward. It declares a float array of size 5, and an integer array of the same size. The float array elements are all initialized to 0.3, and the integer array elements to 1. The program also declares two pointer variables, one a float pointer and the other an integer pointer. Each pointer variable is initialized to point to the array element with index 3; for example, `fptr` is initialized to point to the float array element, `faray[3]`. Therefore, `fptr - 1` points to `faray[2]`, and `fptr + 1` points to `faray[4]`. The value of `*fptr` is then modified, as is the value of `*(fptr - 1)` and `*(fptr + 1)`. Similar changes are made in the integer array. Finally, the arrays are printed. Here is the output of the program:

```
faray[0] = 0.300000 iaray[0] = 1
faray[1] = 0.300000 iaray[1] = 1
faray[2] = 0.900000 iaray[2] = -1
faray[3] = 1.000000 iaray[3] = 0
faray[4] = 1.100000 iaray[4] = 2
```

7.3.2 Array Names vs Pointer Variables

As we have seen, when we declare an array, a contiguous block of memory is allocated for the cells of the array and a pointer cell (of the appropriate type) is also allocated and initialized to point to the first cell of the array. This pointer cell is given the name of the array. When memory is allocated for the array cells, the starting address is fixed, i.e. it cannot be changed during program execution. Therefore, the value of the pointer cell should not be changed. To ensure that this pointer is not changed, in C, array names may not be used as variables on the left of an assignment statement, i.e. they may not be used as an **Lvalue**. Instead, if necessary, separate pointer variables of the appropriate type may be declared and used as **Lvalues**. For example, we can use pointer arithmetic and the dereference operator to initialize an array as follows:

```
/* Use of pointers to initialize an array */
#include <stdio.h>
main()
{   int i;
    float X[MAX];

    for (i = 0; i < MAX; i++)
        *(X + i) = 0.0;      /* same as X[i] */
}
```

In the loop, `*(X + i)` is the same as `X[i]`. Since `X` (the pointer cell) has a fixed value we cannot use the increment operator or the assignment operator to change the value of `X`:

```
X = X + 1;      /* ERROR */
```

Here is an example of a common error which attempts to use an array as an **Lvalue**:

```
/* BUG: Attempt to use an array name as an Lvalue */
#include <stdio.h>
main()
{   int i;
    float X[MAX];

    for (i = 0; i < MAX; i++)  {
        *X = 0.0;
        X++;        /* BUG: X = X + 1; */
    }
}
```

In this example, `X` is fixed and cannot be used as an **Lvalue**; the compiler will generate an error stating that an **Lvalue** is required for the **++** operator. However, we can declare a pointer variable,

which can point to the same type as the type of the array, and initialize it to the value of array pointer. This pointer variable CAN be used as an Lvalue, so we can then rewrite the previous array initialization loop as follows:

```
/*  OK: A pointer variable is initialized to an array pointer and then
    used as an Lvalue.
*/
#include <stdio.h>
main()
{   int i;
    float *ptr, X[MAX];

    ptr = X;       /* ptr is a variable which can be assigned a value */
    for (i = 0; i < MAX; i++) {
        *ptr = 0.0;         /* *ptr accesses X[i] */
        ptr++;
    }
}
```

Observe that the pointer variable, ptr, is type float *, because the array is of type float. It is initialized to the value of the fixed pointer, X (i.e. the initial value of ptr is set to the same as that of X, namely, &X[0]), and may subsequently be modified in the loop to traverse the array. The first time through the loop, *ptr (X[0]) is set to zero and ptr is incremented by one so that it points to the next element in the array. The process repeats and each element of the array is set to 0.0. This behavior is shown in Figure 7.10. Observe that the final increment of ptr makes it point to X[MAX]; however, no such element exists (recall, an array of size MAX has cells indexed 0 to MAX - 1). At the end of the for loop, the value of ptr is meaningless since it now points outside the array. Unfortunately, C does not prevent a program from accessing objects outside an array boundary; it merely increments the pointer value and accesses memory at the new address. The results of accessing the array with the pointer, ptr at this point will be meaningless and possibly disastrous. It is the responsibility of the programmer to make sure that the array boundaries are not breached. The best way of ensuring that a program stays within array boundaries is to write all loops that terminate when array limits are reached. When passing arrays in function calls, always pass the array limit as an argument as well.

Here is a similar error in handling strings and pointers:

```
/* BUG: Common error in accessing strings */
#include <stdio.h>
#define SIZE 100
main()
{   char c, msg[SIZE];

    while ((c = getchar()) != '\n') {
        *msg = c;
```
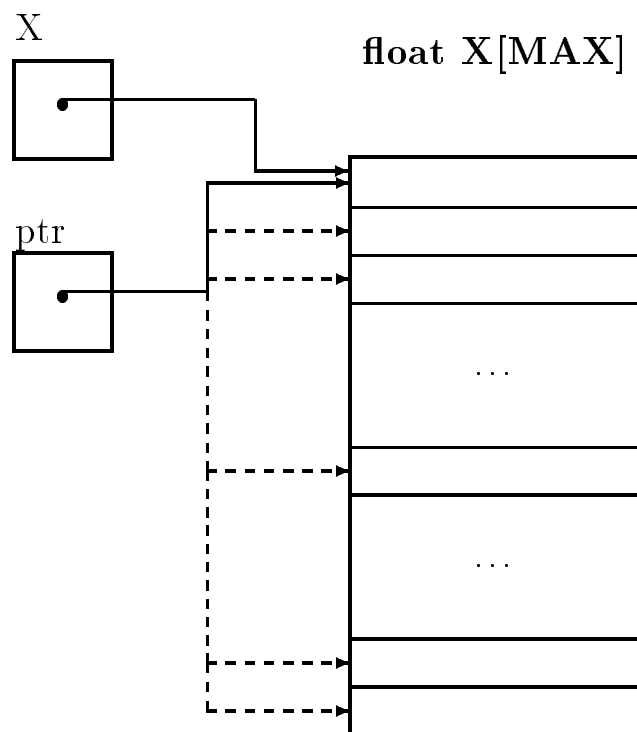
Figure 7.10: Pointer Variables and Arrays

```
      msg++;              /* msg is fixed; it cannot be an Lvalue */
   }
   *msg = '\0';
}
```

The array name, msg is a constant pointer; it cannot be used as an Lvalue. We can rewrite the
loop correctly to read a character string as:

```
   /* OK: Correct use of pointers to access a string */
   #include <stdio.h>
   #define SIZE 100
   main()
   {   char c, *mp, msg[SIZE];

       mp = msg;
       while ((c = getchar()) != '\n') {
             *mp = c;
             mp++;              /* mp is a variable; it can be an Lvalue */
       }
       *mp = '\0';
   }
```
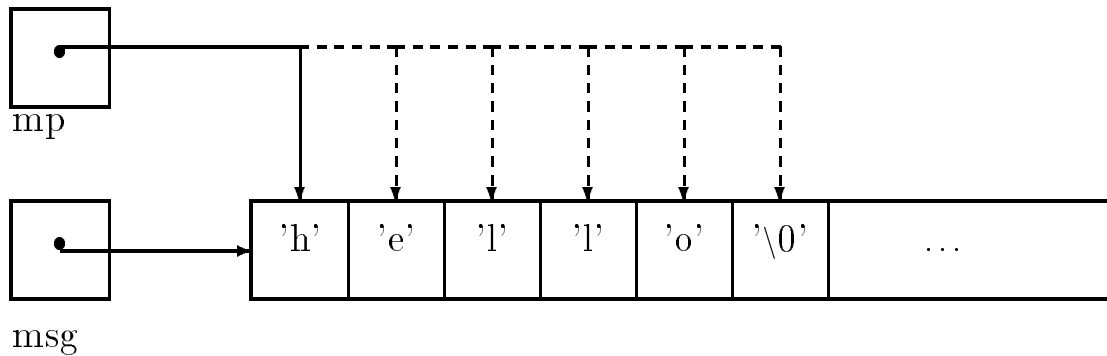
Figure 7.11: Pointer Variables and Strings

Observe in this case, `mp` is a character pointer since the array is a character array. The variable, `mp` is initialized to the value of `msg`. The dereferenced pointer variable, `*mp`, then accesses the elements of the array in sequence as `mp` is incremented (see Figure 7.11). The loop terminates when a newline is read, and a terminating `NULL` is added to the string.

Remember, pointer variables must be initialized to point to valid objects; otherwise, fatal errors will most likely occur. For example, if the pointer, `mp`, in the above code were not initialized to the value of `msg`, a serious and probably fatal error will occur when the pointer is dereferenced and an attempt is made to access the memory cell pointed to by `mp`. This is because the initial value of `mp` would be some garbage value which may point to an invalid memory address causing a fatal memory fault to occur. If the garbage value were not an invalid memory address, the loop would write characters to an unknown memory address, possibly destroying other valid data.

As we've said, an array names cannot be used as an Lvalues. On the other hand, when a function is used to access an array, the corresponding formal parameter is a pointer variable. This pointer variable can be used as an Lvalue. Here is a function to print a string:

```
/* Function prints a string pointed to by mp. */
void our_strprint(char *mp)
{
    while (*mp) {
        putchar(*mp);
        mp++;              /* mp is a variable; it can be an Lvalue */
    }
    putchar('\n');
}
```

Here, `mp` is a pointer variable, which, when the function is called, we assume will be initialized to point to some `NULL` terminated string. The expression, `*mp`, accesses the elements of the array, and the loop continues as long as `*mp` is not `NULL`. Each time the loop is executed, a character, `*mp`, is written, and `mp` is incremented to point to the next character in the array. When `*mp` accesses the `NULL`, the loop terminates and a newline character is written.

7.4 String Assignment and I/O

As we have seen, a character string in C is an array of characters with a terminating NULL character. Access to a character string requires only a pointer to the character array containing the characters. It is common to use the term, **string**, to loosely refer to either an array of characters holding the string, or to a character pointer that may be used to access the string; it should be clear from context which is meant.

When a character *string constant* is used in a program, the compiler automatically allocates an array of characters, stores the string in the array, appends the NULL character, and replaces the string constant by the value of a pointer to the string. Therefore, the value of a string constant is the value of a pointer to the string. We can use string constants in expressions just as we can use the names of arrays. Here is an example:

```
char *mp, msg[SIZE];

mp = "This is a message\n";
```

The compiler replaces the string constant by a pointer to a corresponding string. Since `mp` is a character pointer variable, we can assign a value of a fixed string pointer to `mp`. If necessary we can traverse and print the string using this pointer. On the other hand, since `msg[]` is declared as a character array, we cannot make the following assignment:

```
msg = "This is a message\n";        /* ERROR */
```

since we are attempting to modify a *constant* pointer, `msg`.

A string constant is just another string appropriately initialized and accessed by a pointer to it. We will therefore make no distinctions between strings and string constants; they are both strings referenced by string pointers. While strings and string constants are both strings, the contents of string constants cannot be changed in ANSI C.

We have been using string constants as format strings for `printf()` and in `scanf()`, which expect their first argument to be a string pointer; i.e. a `char` pointer. The compiler has automatically created an appropriate string and replaced the string by a string pointer. Instead of writing a format string directly in a function call, we could pass a string pointer that points to a format string. Here is an example:

```
char *mesg;
int n;

n = 1;
mesg = "This is message number %d\n";
printf(mesg, n);
```

The string constant is stored by the compiler somewhere in memory as an array of characters with an appended `NULL` character. A pointer to this character array is assigned to the character pointer variable, `mesg`. The function `printf()` then uses the pointer to retrieve the format string, and print the string:

```
This is message number 1
```

The functions, `printf()` and `scanf()` can be used for string input and output as well. Array names or properly initialized pointers to strings must be passed as arguments in both cases. The conversion specification for strings is `%s`. For example, consider the task of reading strings and writing them out. Here is an example program.

```
/*  File: fcopy.c
    This program reads strings from standard input using scanf() and writes
    them to standard output using printf().
*/
#include <stdio.h>
#include "araydef.h"
main()
{   char mesg[SIZE];

    printf("***Strings: Formatted I/O***\n\n");
    printf("Type characters, EOF to quit\n");
    while (scanf("%s", mesg) != EOF)
        printf("%s\n", mesg);
}
```

Sample Session:

```
***Strings:  Formatted I/O***

Type characters, EOF to quit
        This        is        a        test
This
is
a
test
^D
```

The conversion specification, `%s` indicates a string and the corresponding matching argument must be a char pointer. When `scanf()` reads a string it stores it at the location pointed to by `mesg` — note we do not use `&mesg` since `mesg` is already a pointer to an array of characters. Then, `printf()` prints the string pointed to by `mesg`. When `scanf()` reads a string using `%s`, it behaves like it

does for numeric input, skipping over leading white space, and reading the string of characters until a white space is reached. Thus, `scanf()` can read only single words, storing the string of characters read into an array pointed to by the argument, `mesg` and appending a `NULL` character to mark the end of the string. On the other hand, `printf()` prints the string pointed to by its argument, `mesg`, printing the entire string (including any white space) until a `NULL` character is reached. The sample session shows that each time `scanf()` reads a string, only a single word is read from the input line and then printed.

As we said, when `scanf()` reads a string, the string argument must be a pointer that points to an array where the input characters are to be stored. For example, here are correct and incorrect ways of using `scanf()`:

```
char * mp, * mptr, msg[SIZE];

scanf("%s", mp);     /* BUG */
scanf("%s", msg);    /* OK */
mptr = msg;
scanf("%s", mptr);   /* OK */
```

The first `scanf()` is incorrect because `mp` has not been initialized and, therefore, does not point to an array where a string is to be stored. The other statements are correct; in each case, the pointer points to an array.

7.5 Array Initializers

ANSI C allows automatic array variables to be initialized in declarations by constant initializers as we have seen we can do for scalar variables. These initializing expressions must be *constant* (known at compile time) values; expressions with identifiers or function calls may not be used in the initializers. The initializers are specified within braces and separated by commas. Here are some declarations with constant initializers:

```
int ex[10] = { 12, 23, 9, 17, 16, 49 };
char word[10] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

Each constant initializer in the braces is assigned, in sequence, to an element of the array starting with index 0. If there are not enough initializers for the whole array, the remaining elements of the array are initialized to zero. Thus, `ex[0]` through `ex[5]` are assigned the values 12, 23, 9, 17, 16, and 49, while `ex[6]` through `ex[9]` are initialized to zero. Similarly, `word` is initialized to a string `"hello"`. String initializers may be written as string constants instead of character constants within braces, for example:

```
char mesg[] = "This is a message";
char name[20] = "John Doe";
```

In the case of `mesg[]`, enough memory is allocated to accommodate the string plus a terminating
`NULL`, and we do not need to specify the size of the array. The above string initializers are allowed
as a convenience, the compiler initializes the array at compile time. Remember, initializations
are *not* assignment statements; they are declarations that allocate and initialize memory. As with
other arrays, these array names cannot be used as `Lvalues` in assignment statements.

Here is a short program that shows the use of initializers:

```
/* File: init.c
   Program shows use of initializers for arrays.
*/
#include <stdio.h>
#define MAX 10
main()
{   int i, ex[MAX] = { 12, 23, 9, 17, 16, 49 };
    char word[MAX] = {'S', 'm', 'i', 'l', 'e', '\0'};
    char mesg[] = "Message of the day is: ";

    printf("***Array Declarations with Initializers***\n\n");
    printf("%s%s\n", mesg, word);
    printf("Initialized Array:\n");
    for (i= 0; i < MAX; i++)
        printf("%d\n", ex[i]);
}
```

Sample Output:

```
***Array Declarations with Constant Initializers***

Message of the day is:  Smile
12
23
9
17
16
49
0
0
0
0
```

The first `printf()` statement uses `%s` to print each of the two strings accessed by pointers, `mesg`
and `word`. Finally, the loop prints the array, `ex`, one element per line.
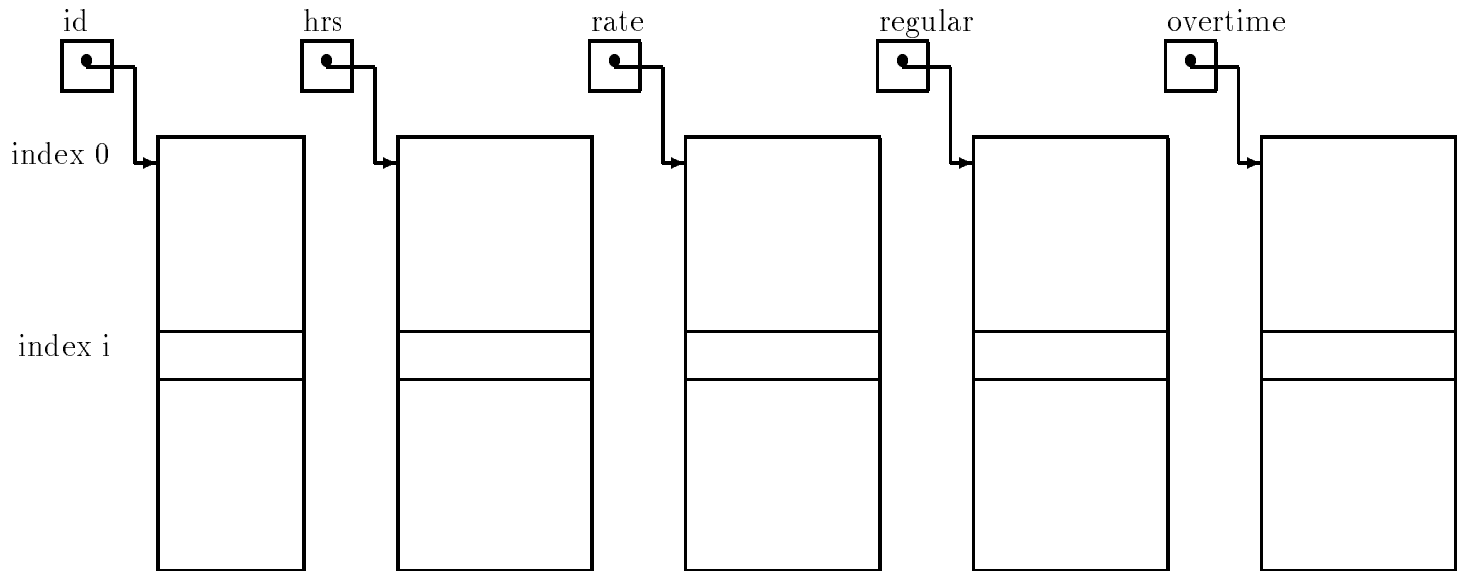
Figure 7.12: Data Record Spread Over Several Arrays

7.6 Arrays for Databases

We now consider our payroll task that reads input data and calculates pay as before, but the program prints a table of computed pay for all the id's. The algorithm uses arrays to store the data, but is otherwise very similar to our earlier programs: get data, calculate pay, and print results. We will use functions to perform these subtasks. Here are the prototypes:

```
/* File: payutil.h */
int getdata(int id[], float hrs[], float rate[], int lim);
void calcpay(float hrs[], float rate[], float reg[], float over[], int n);
void printdata(int id[], float hrs[], float rate[],
                        float reg[], float over[], int n);
```

The function, `getdata()` gets the data into the appropriate arrays for id's, hours worked, and rate of pay; returning the number of id's entered by the user. While the arrays `id[]`, `hrs[]`, and `rate[]` are individual arrays, we make sure that the same value of the array index accesses the data for a given id. For example, `id[i]` accesses an id number and `hrs[i]` and `rate[i]` access hours worked and rate of pay for that id number. In other words, an input *data record* for each id number resides at the same index in these arrays.oWe can think of this *data structure* as a table, where the columns are the arrays holding different pieces of information; and the rows are the data for an individual id, as shown in Figure 7.12.

Next, `calcpay()` calculates and stores regular and overtime pay for each id in arrays, `regpay[]` and `overpay[]` (columns), at the same array index as the input data record. Thus, the entire payroll data record for each id number is at a unique index in each of the arrays. Finally, `printdata()`

```
/*  File: paytab.c
    Other Source Files: payutil.c
    Header FIles: paydef.h, payutil.h
    Program calculates and stores payroll data for a number of id's. Gets
    data, calculates pay, and prints data for all id's.
*/
#include <stdio.h>
#include "paydef.h"
#include "payutil.h"
#define MAX 10

main()
{   int n, id[MAX];
    float hrs[MAX], rate[MAX], regpay[MAX], overpay[MAX];

    printf("***Payroll Program***\n\n");
    n = getdata(id, hrs, rate, MAX);
    calcpay(hrs, rate, regpay, overpay, n);
    printdata(id, hrs, rate, regpay, overpay, n);
}
```

Figure 7.13: Code for `paytab.c`

prints each payroll record, i.e. the input data as well as the calculated regular, overtime, and total pay. We will write `getdata()`, `printdata()`, and `calcpay()` in the file, `payutil.c`. The prototypes shown above for these functions are in the file, `payutil.h`. This header file is included in the program file `paytab.c`, where `main()` will reside (see Figure 7.13). We also include the header file, `paydef.h` which defines the symbolic constants, `REG_LIMIT` and `OT_FACTOR.`:

```
/* paydef.h */
#define REG_LIMIT 40
#define OT_FACTOR 1.5
```

The program calls `getdata()` which reads data into the appropriate arrays and stores the returned value (the number of id's) into `n`. It then calls on `calcpay()` to calculate the pay for `n` people, filling in the `regpay[]` and `overpay[]` arrays, and calls `printdata()` to print the input data and the results for `n` people. The code for these functions is shown in Figure 7.14.

In the function, `getdata()`, `scanf()` is used to read data for the itemsa, using `n` to count and index the data items in the arrays. We use pointer arithmetic to pass the necessary arguments to `scanf()`. For example, to read data into `id[n]`, we must pass its address `&id[n]`. Instead, we pass `id + n` which is identical to `&id[n]`. The function, `getdata()`, reads data for as many id's as possible, returning either when there is no more data (a zero id value) or the arrays are full (`n` reaches the limit, `lim` passed in). If the array limit is reached, an appropriate message is printed

```
/* File: payutil.c */
#include <stdio.h>
#include "paydef.h"
#include "payutil.h"
/* Gets data for all valid id's and returns the number of id's */
int getdata(int id[], float hrs[], float rate[], int lim)
{   int n = 0;
    while (n < lim) {
        printf("ID <zero to quit>: ");
        scanf("%d", id + n);            /* id + n  is same as &id[n] */
        if (id[n] <= 0) return n;
        printf("Hours Worked: ");
        scanf("%f", hrs + n);           /* hrs + n is same as &hrs[n] */
        printf("Rate of Pay: ");
        scanf("%f", rate + n);          /* rate + n is same as &rate[n] */
        n++;
    }
    printf("No more space for data - processing data\n");
    return n;
}
/* Calculates regular and overtime pay for each id */
void calcpay(float hrs[], float rate[], float reg[], float over[], int n)
{   int i;
    for (i = 0; i < n; i++) {
        if (hrs[i] <= REG_LIMIT) {
            reg[i] = hrs[i] * rate[i];
            over[i] = 0;
        }
        else {
            reg[i] = REG_LIMIT * rate[i];
            over[i] = (hrs[i] - REG_LIMIT) * OT_FACTOR * rate[i];
        }
    }
}


/* Prints a table of payroll data for all id's. */
void printdata(int id[], float hrs[], float rate[],
                       float reg[], float over[], int n)
{   int i;
    printf("***PAYROLL: FINAL REPORT***\n\n");
    printf("%4s\t%5s\t%5s\t%6s\t%6s\t%6s\n", "ID", "HRS", "RATE",
            "REG", "OVER", "TOT");
    for (i = 0; i < n; i++)
        printf("%4d\t%5.2f\t%5.2f\t%6.2f\t%6.2f\t%6.2f\n",
                id[i], hrs[i], rate[i], reg[i], over[i],
                reg[i] + over[i]);
}
```

Figure 7.14: Code for payutil.c

and the input is terminated. The function returns the number of id's placed in the arrays. The other functions in the program are straight forward; each index accesses the data record for the id at that index.

As written, `getdata()` terminates input of data when an invalid id (`id <= 0`) is entered. An alternative might be to read a data item in a temporary variable first, examine it for validity if desired, and then assign it to an array element. For example:

```
scanf("%d", &x);
if (x > 0)
    id[n] = x;
else
    return n;
```

Here is a sample session for the program, `paytab.c` compiled and linked with `payutil.c`:

Sample Session:

```
***Payroll Program***

ID <zero to quit>:  8
Hours Worked:  50
Rate of Pay:  14
ID <zero to quit>:  12
Hours Worked:  45
Rate of Pay:  12.50
ID <zero to quit>:  2
Hours Worked:  20
Rate of Pay:  5
ID <zero to quit>:  5
Hours Worked:  40
Rate of Pay:  10
ID <zero to quit>:  0
***PAYROLL: FINAL REPORT***


ID    HRS    RATE     REG     OVER     TOT

 8  50.00  14.00  560.00  210.00  770.00
12  45.00  12.50  500.00   93.75  593.75
 2  20.00   5.00  100.00    0.00  100.00
 5  40.00  10.00  400.00    0.00  400.00
```

7.7   Common Errors

1. Use of an array name as an **Lvalue**: An array name has a fixed value of the address where
   the array is allocated. It is NOT a variable; it cannot be used as an **Lvalue** and assigned a
   new value. Here are some example:

   (a) `int x[10];`

   ```
   while (*x) {
       ...
       x++;        /* ERROR */
   }
   ```
   `x` cannot be used as an **Lvalue** and assigned new values.

   (b) `char msg[80];`
   ```
   ...
   while (*msg) {
       ...
       msg++;     /* ERROR */
   }
   ```
   `msg` cannot be used as an **Lvalue**.

   (c) `char msg[80];`

   ```
   msg = "This is a message";   /* ERROR */
   ```
   `msg` cannot be an **Lvalue**. The right hand side is not a problem. Value of a string
   constant is a pointer to an array automatically allocated by the compiler.

   (d) `char msg[80] = "This is a message";`

   ```
   /* OK: array initialized to the string when memory is allocated */
   ```
   A string constant initializer is correct. When memory is allocated for the array, it is
   initialized to the string constant specified.

2. Failure to define an array: Definition of an array is required to allocate memory for storage
   of an array of objects. A pointer type allocates memory for a pointer variable, NOT for
   an array of objects. Suppose, `read_str()` reads a string and stores it where its argument
   points:

   ```
   int *pmsg;    /* memory allocated for a pointer variable */

   read_str(pmsg);    /* ERROR: memory not allocated for a string */
   ```

   No memory is allocated for a string, i.e. an array of characters. The variable, `pmsg` points to
   some garbage address; `read_str()` will attempt to store the string at that garbage address.
   The address may be invalid, in which case there will be a memory fault; a fatal error.
   Allocate memory for a string with an array declaration:

```
int str[MAX];
read_str(str);
```

3. Array pointer not passed to a called function: If a called function is to store values in an array for later use by the calling function, it should be passed a pointer to an array defined in the calling function. Here is a program with an error.

```
#include <stdio.h>
main()
{   char * p, s[80],
        * get_word(char * s);

    p = get_word(s); /* ERROR: returned pointer points to freed memory */
    puts(p);            /* Prints garbage */
}

char * get_word(char *str)
{   char wd[30];            /* memory allocated for array wd[] */
    int i = 0;
    while (*str == ' ')      /* skip leading blanks */
        ;
    while (*str != ' ')      /* while not a delimiter */
        wd[i++] = *str++;    /* copy char into array wd[] */
    wd[i] = '\0';            /* append a NULL to string in wd[] */
    return wd;               /* return pointer to wd[] */
}                            /* memory for array wd[] is freed */
```

The function, get_word() copies a word from a string, s, into an automatic array, wd[] for which memory is allocated in get_word(). When get_word() returns, a pointer, wd, to the calling function, the memory allocated for wd[] is freed for other uses, since it was allocated only for get_word(). The data stored in wd[] may be overwritten with other data. In the calling function, p is assigned an address value which points to freed memory. The function, puts(), will print a garbage sequence of characters pointed to by p. At times, the memory may not be reused right away and it will print the correct string. At other times, it will print out garbage.

4. Errors in passing array arguments: Only array names, i.e., pointers to arrays, should be passed as arguments. The following are all in ERROR:

```
func(s[]);
func(s[80]);
func(*s);
```

Pointers to arrays, i.e. array names by themselves, should be passed as arguments in function calls. Arguments in the above function calls are not pointers. The first one is meaningless in an expression; the second attempts to pass an element at index 80; the third passes a dereferenced pointer, not the pointer to the array.

5. Errors in declaring formal parameters: Formal parameters referencing arrays in function definitions should be specified to be pointers, not objects of a base type. Consider a function, `init()`, that initializes elements of an integer array to some values. The following is an error:

```
init(int aray)
{
        ...
}
```

The parameter declared is an integer not a pointer to an integer. It should be either of the following:

```
init(int * aray)
```

OR

```
init(int aray[])
```

In either of the above cases, memory for an integer *pointer* is allocated, NOT for a new array of integers.

6. Misinterpreting formal parameter declarations: Even if an array size is specified in a formal parameter, memory is not allocated for an array but for a pointer.

```
init(int aray[10])
```

The above declares `aray` as an integer pointer.

7. Pointers are not initialized:

```
 int x, * px;
 x = 10;
 printf("*px = %d\n", *px);
```

Since value of `px` is garbage, there will be a fatal memory fault when an attempt is made to evaluate `*px`.

7.8 Summary

In this Chapter, we have introduced one form of compound data type: the **array**. An array is a block of a number of data items all of the same type allocated in contiguous memory cells. We have seen that, in C, an array may be declared using the syntax:

<type-specifier><identifier>[<size>];

specifying the type of the data items, and the number of elements to be allocated in the array. As we saw, such a declaration in a function causes <size> data items of type <type-specifier> to be allocated in contiguous memory, AND a pointer cell to be allocated of type <type-specifier>* (pointer to <type-specifier>), given the name, <identifier>, and initialized to point to the first cell of the array. More specifically, for a declaration like:

```
int data[100];
```

allocates 100 `int` cells, and an `int *` cell named `data` which is initialized to point to the block of integers.

We saw that the data items in an array can be accessed using an **index**; i.e. the number of the item in the block. Numbering of data items begins with index 0, to the size - 1. We use the index of an element in a **subscripting** expression with syntax:

```
<identifier>[<expression>]
```

where <identifier> is the name of the array, and the <expression> in the square brackets ([]) is evaluated to the index value. So, for our previous example, the statement:

```
data[0] = 5;
```

sets the integer value, 5, into the first cell of the array, `data`. While:

```
data[i] = data[i-1];
```

would copy the value from the element with index `i - 1` to its immediate successor in the array.

The data types of the elements of an array may be any scalar data type; `int`, `float`, or `char`. (We will see other types for array elements in later chapters). We have cautioned that, in C, no checking is done on the subscripting expressions to ensure that the index is withing the block of data allocated (i.e. that the subscript is *in bounds*). It is the programmers responsibility to ensure the subscript is in bounds. We have seen two ways of doing this: to keep the value of the limit or the extent of data in the array in a separate integer variable and perform the necessary comparisons, or to mark the last item in the array with a special value. The most common use of this later method is in the case of an array of characters (called a **string**), where the end of the string is indicated with the special character, `NULL` (whose value is 0).

We have also discussed the equivalence of subscripting expressions and pointer arithmetic; i.e. that a subscripting expression, `data[i]`, is equivalent to (and treated by the compiler as) the pointer expression, `*(data + i)`. Remember, the *name* of the array is a pointer variable, pointing to the first element of the array. These two forms of array access may be used interchangebly in programs, as fits the logic of the operation being performed. It is the semantics of pointer arithmetic that will compute the address of the indexed element correctly.

In addition, we have seen that passing arrays as parameters to functions is done by passing a pointer to the array (the array name). The cells of data are allocated in the calling function, and the called function can access them indirectly using either a pointer expression or a subscripting expression. Remember, a parameter like:

```
int func( int a[] )
```

(even if it has a <size> in the brackets) does NOT allocate integer cells for the array; it merely allocates an `int *` cell which will be given a pointer to an array in the function call. Such a parameter is exactly equivalent to:

```
int func( int *a)
```

We have discussed the fact that the pointer cell, referenced using the name of the array, is a *constant* pointer cell; i.e. it may not be changed by the program (it may not be used as an `Lvalue`). However; additional pointer cells of the appropriate type may be declared and initialized to point to the array (by the programmer) and can then be used to traverse the array with pointer arithmetic (such as the `++` or `--` operators).

We have shown how arrays can be initialized in the declaration (a bracketed, comma separated list of values, or, for strings, a string constant). We have seen the semantics of string assignment and how strings can be read and printed by `scanf()` and `printf()` using the `%s` conversion specifier. Remember, for `scanf()`, `%s` behaves like the numeric conversion specifiers; it skips leading white space and terminates the string (with a `NULL`) at the first following white space character.

Finally, we have shown an example of using arrays in a data base type applications, where arrays of different types were used to hold a collection of payroll records for individuals. In that example, the elements at a specific index in all of the arrays corresponded to one particular data record.

The *array* is an important and powerful data structure in any programming language. Once you master the use of arrays in C, the scale and scope of your programming abilities expand tremendously to include just about any application.

7.9   Exercises

With the following declaration:

```
int *p, x[10];
char *t, s[100];
```

Explain each of the following expressions. If there is an error, explain why it is an error.

1. (a)   `x`

   (b)   `x + i`

   (c)   `*(x + i)`

   (d)   `x++;`

2. (a)   `p = x;`

   (b)   `*p`

   (c)   `p++;`

   (d)   `p++;`

   (e)   `p--;`

   (f)   `--p;`

3. (a) `p = x + 5;`

   (b)   `*p;`

   (c)   `--p;`

   (d)   `p*;`

4. `scanf("%s", s);`

   Input: Hello, Hello.

5.   `printf("%s\n", s);`

6.   `scanf("%s", t);`
       `t = s;`
       `scanf("%s", t);`

   Check the following problems; find and correct errors, if any. What will be the output in each case.

```
 7. main()
    {   int i, x[10] = { 1, 2, 3, 4};

        for (i = 0; i < 10; i++) {
            printf("%d\n", *x);
            x++;
        }
    }

 8. main()
    {   int i, *ptr, x[10] = { 1, 2, 3, 4};

        for (i = 0; i < 10; i++) {
            printf("%d\n", *ptr);
            ptr++;
        }
    }

 9. main()
    {   int i, x[10] = { 1, 2, 3, 4};

        for (i = 0; i < 10; i++)
            printf("%d\n", (x + i));
    }

10. main()
    {   int i, x[10] = { 1, 2, 3, 4};

        for (i = 0; i < 10; i++)
            printf("%d\n", *(x + i));
    }

11. main()
    {   int i, *ptr, x[10] = {1, 2, 3, 4};

        ptr = x;
        for (i = 0; i < 10; i++) {
            printf("%d\n", *ptr);
            ptr++;
        }
    }

12. main()
    {   int i, *ptr, x[10] = {1, 2, 3, 4};

        ptr = x;
        for (i = 0; i < 10; i++) {
            printf("%d\n", ptr);
```

```
            ptr++;
      }
}
```

13.
```
main()
{   char x[10];

    x = "Hawaii;
    printf("%s\n", x);
}
```

14.
```
main()
{   char *ptr;

    ptr = "Hawaii";
    printf("%s\n", ptr);
}
```

15.
```
main()
{   char *ptr, x[10] = "Hawaii";

    for (i = 0; i < 10; i++)
        printf("%d %d %d\n", x + i, *(x + i), x[i]);
}
```

16.
```
main()
{   char x[10];

    scanf("%s", x);
    printf("%s\n", x);
}
```

The Input is:

```
        Good Day to You
```

17.
```
main()
{   char *ptr;

    scanf("%s", ptr);
    printf("%s\n", ptr);
}
```

The Input is:

```
        Good Day to You
```

18. Here is the data stored in an array

```
char s[100];
```

```
Hawaii\0Manoa\0
```

What will be printed out by the following loop?

```
i = 0;
while (s[i]) {
    putchar(s[i]);
    i++;
}
```

7.10   Problems

1. Write a program that uses the sizeof operator to print the size of an array declared in the program. Use the sizeof operator on the name of the array.

2. Write a function that prints, using dereferenced pointers, the elements of an array of type float.

3. Write a function that checks if a given integer item is in a list. Traverse the array and check each element of the list. If an element is found return True; if the array is exhausted, return False.

4. Write a function that takes an array of integers and returns the index where the maximum value is found.

5. Write a function that takes an array and finds the index of the maximum and of the minimum.

   Use arrays to house sets of integers. A set is a list of items. Each item of a list is a member of a list and appears once and only once in a list. Write the following set functions.

6. Test if a number is a member of a set: is the number present in the set?

7. Union of two sets A and B: the union is a set that contains members of each of the two sets A and B.

8. Intersection of two sets A and B: the intersection contains only those members that are members of both the sets A and B.

9. Difference of two sets A and B: The new set contains elements that are members of A that are not also members of B.

10. Write a function to read a string from the standard input. Read characters until a newline is reached, discard the newline, and append a NULL. Use array indexing.

11. Write a function to read a string from the standard input. Read characters until a newline is reached, discard the newline, and append a NULL. Use pointers.

12. Write a function to write a string to the standard output. Write characters until a NULL is reached, discard the NULL, and append a newline. Use pointers.

13. Write a function to change characters in a string: change upper case to lower case and vice versa. Use array indexing.

14. Write a function to change characters in a string: change upper case to lower case and vice versa. Use pointers.

15. Write a function that counts and returns the number of characters in a string. Do not count the terminating NULL. Use array indexing.

16. Write a function that counts and returns the number of characters in a string. Do not count the terminating NULL. Use array indexing.

17. Write a function that removes the last character in a string. Use array indexing to reach the last element and replace it with a NULL.

18. Write a function that removes the last character in a string. Use pointers to traverse the array.

19. Repeat problems 24 and 25, but use the function of problems 22 or 23.

Chapter 8

Functions and Files

In this Chapter, we tie up some loose ends concerning some of the built in functions provided by the C language. In previous chapters we have been using such functions in our programming examples to do data input and output; functions such as `scanf()`, `printf`, `getchar()`, and `putchar()`. These routines are part of a library of standard routines. As we have seen, we can use these functions by including the header file in which they are declared (in this case <`stdio.h`>). These header files contain the prototypes for functions as well as macros that are needed for their use.

Previously, when we have needed routines for other operations (e.g. testing if a character is a digit), we have written our own. Such operations are common enough in C programs that the implementors have included predefined routines to perform them. These routines are collectively called the C Standard Library. We begin this Chapter by describing a few other built in functions provided in the Standard Library, describing their use and using them in a few sample programs. A longer (though not complete) listing of the Standard Library, together with descriptions, is provided in Appendix C.

We next give a more thorough description of our I/O functions, `scanf()` and `printf()`. Finally, we discuss variations of the standard I/O routines, which allow direct access to data stored in files.

8.1   The C Standard Library

We have already used several I/O routines from the standard library: scanf(), printf(), getchar() and putchar(). Many other useful routines are provided in one or more libraries supplied with the compiler or in header files. When a function in one of these libraries is used, the name of the library must be supplied to the linker. Otherwise, the linker is unable to resolve the reference to that function. If the function resides in the standard library, the linker does not need to be supplied the name. The linker always searches the standard library by default for any unresolved functions used in the program.

Standard header files supplied with the compiler declare function prototypes for standard library functions in several categories. They also define data types, symbolic constants, and macros. Header files must be included in the source program if any of the definitions, macros, or function prototypes declared in them are to be used.

Many of the functions we have defined in our example programs are available either as standard macros in a header file or as functions in the standard library. We could have used these standard

routines in many of our examples. However, we wrote our own versions because it is instructive to see how functions are written.

The following are descriptions of some of the commonly used routines. Similar descriptions of other routines will be provided as we use them. A listing of ANSI standard library routines is provided in Appendix C. It must be understood that the standard is a suggested standard, and all vendors of C compilers may not follow the suggested standard exactly.

The listing below specifies what header file must be included, if any, before the routine listed may be used. It also specifies which file contains the prototypes, if applicable.

8.1.1   Character Processing Routines

Character Classification Routines

is...   *Prototype:*

```
int isalnum(int c);
int isalpha(int c);
int isascii(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

in: `<ctype.h>`

Returns:

isalnum	TRUE if c is a letter or a digit. ('A'-'Z', 'a'-'z', '0'-'9').
isalpha	TRUE if c is a letter. ('A'-'Z', 'a'-'z')
isascii	TRUE if c is in the range 0-127.
iscntrl	TRUE if c is a control character. (0-31, 127)
isdigit	TRUE if c is a digit. ('0'-'9')
isgraph	TRUE if c is a graphical character, i.e. a printable character except for space. (33-126)
islower	TRUE if c is a lower case letter. ('a'-'z')
isprint	TRUE if c is a printable character. (32-126)
ispunct	TRUE if c is a punctuation character.
isspace	TRUE if c is a space, tab, newline, or any white space character. (9-13, 32)
isupper	TRUE if c is an upper case letter. ('A'-'Z')
isxdigit	TRUE if c is a hexadecimal digit. ('0'-'9', 'A'-'F', 'a'-'f')

Description: These are macros that classify a character, c, given as an integer type (ASCII) value. They return non-zero if TRUE and zero if FALSE.

Character Conversion Routines

toascii   *Prototype:*   `int toascii(int c);`   *in:* `<ctype.h>`

                Returns: converted value of `c`.

                Description: Converts an integer, `c`, to ASCII format by clearing all but the lower seven bits. The value returned is in the range 0-127.

tolower   *Prototype:*   `int tolower(int c);`   *in:* `< ctype.h>`

                Returns: Lower case value of `c` if `c` was upper case, `c` otherwise.

toupper   *Prototype:*   `int toupper(int c);`   *in:* `< ctype.h>`

                Returns: Converts `c` to upper case if `c` is lower case; otherwise, `c` is left unchanged.

Note that all the above library character routines use an `int` type argument. Since the value of a character is its ASCII value of type `int`, passing a `char` type argument to these routines is the same as passing an `int` type ASCII value.

Character Routines Programming Examples

Let us use some of the above library routines to write a variation on our previous program to pick out words in the input text. The revised program only picks out valid words; namely identifiers. We will assume that a valid identifier starts with a letter and may be followed by any number of letters and/or digits. White space delimits an identifier; otherwise, it is ignored. Any character that does not belong in an identifier is an illegal character; and also delimits an identifier.

We will need to test each character to see if it is a letter, a digit, a white space, etc. We will use library functions `isalpha()`, `isalnum()`, and `isspace()` to test for these characters. The descriptions for them states that we must include file `<ctype.h>`. In addition to finding and printing identifiers, the program also keeps a count of them.

The only change in the previous algorithm is that now we start a word if and only if it starts with a letter. Once a word is started, it continues as long as characters are letters or digits; otherwise, the word is terminated and counted. The program is shown in Figure 8.1.

We test if the first character after white space is a letter. If so, we build an identifier. Otherwise, if it is `EOF`, we terminate the loop. Otherwise, it must be an illegal character.

Sample Session:

```
***Print Identifiers***

Type text, terminate with EOF
Programming is easy
Programming
is
easy
once an algorithm is developed
once
an
algorithm
```

```
/*    File: ident.c
      Program reads characters one at a time until EOF. It prints out
      each identifier in the input text and counts the total number of
      identifiers. It ignores white space except as a delimiter for an
      identifier. An identifier starts with an alphabetic letter and may be
      followed by any number of letters or digits. All other characters are
      considered illegal.
*/

#include <stdio.h>
#include <ctype.h>

main()
{     int cnt = 0;
      signed char c;

      printf("***Print Identifiers***\n\n");
      printf("Type text, terminate with EOF ^Z or ^D)\n");
      c = getchar();
      while (c != EOF) {
            while (isspace(c))              /* skip leading white space   */
                  c = getchar();

            if (isalpha(c)) {         /* if a word starts with a letter */
                  while (isalnum(c)) {    /* while c is letter or digit */
                        putchar(c);         /* print c                    */
                        c = getchar();      /* read next char             */
                  }
                  putchar('\n');          /* end identifier with a newline */
                  cnt++;                    /* increment cnt              */
            }

            else if (c == EOF)            /* if end of file             */
                  break;                    /* break out of loop          */

            else {                    /* otherwise, it is an illegal char */
                  printf("Illegal character %c\n", c);
                  c = getchar();
            }
      }
      printf("Number of Identifiers = %d\n", cnt);
}
```

Figure 8.1: Code for ident.c

```
is
developed
^D
Number of Identifiers = 8
```

8.1.2 Math Routines

There are many mathematical routines in the standard library, such as `abs()`, `pow()`, `sqrt()`, `rand()`, `sin()`, `cos()`, `tan()`, and so forth. The prototypes for these are defined in the header file, `<math.h>`, which must be included whenever these functions are used in a program. In addition, on Unix systems, the math library is maintained separately from the standard library, thus requiring that it be linked when the code is compiled. This can be done with the compiler command:

```
cc  filename.c -lm
```

The option `-l` specifies that a library must be linked with the code and the `m` specifies the math library. Note that this option *MUST* appear as the last item on the command line.

Most of the functions listed above are self explanatory (and are described in detail in Appendix C). As an example, let us look at the function, `rand()` which generates pseudo-random integers in the range of numbers from 0 to the largest positive integer value. The numbers cannot be completely random because the range is limited. However, for the most part, the numbers generated by `rand()` appear to be quite random. The prototype for the function is:

```
int rand(void);
```

Each time the function is called, it returns a random integer number. Figure 8.2 shows an example which generates and prints some random numbers.

Sample Session:

```
346
130
10982
1090
11656
```

The random number generator will always start with the same number unless it is "seeded" first by calling the function, `srand()`. The prototype for it is:

```
void srand(unsigned x);
```

In the example in Figure 8.3, we seed the random number generator with a user supplied number. The program then finds random throws for a single dice. After the random generator is seeded, every random number generated, `n`, is evaluated modulo 6, i.e. `n % 6` is evaluated. This results in numbers from 0 to 5. We add one to obtain the dice throws from 1 to 6.

Sample Session:

```
***Single Dice Throw Program***
```

```
/* File: rand.c
      Program uses random number generator to print some random
      numbers.
*/
#include <stdio.h>
#include <math.h>
main()
{     int i;
      int x;

      for (i = 0; i < 5; i++) {
            x = rand();
            printf("%d\n", x);
      }
}
```

Figure 8.2: Small program to generate random numbers

```
/* File: dice.c
      Program throws a single dice repeatedly.
*/
#include <stdio.h>
#include <math.h>

main()
{     int i, d1;

      printf("***Single Dice Throw Program***\n\n");
      printf("Type a random unsigned integer to start: ");
      scanf("%d", &i);
      srand(i);

      for (i = 0; i < 5; i++) {
            d1 = rand() % 6 + 1;
            printf("throw = %d\n", d1);
      }
}
```

Figure 8.3: Program for generating random dice values

```
Type a random unsigned integer to start:   12737
throw = 2
throw = 6
throw = 1
throw = 3
throw = 5
```

Similarly, we can write a program that draws a card from a full deck of 52 cards as shown in Figure 8.4. It starts by seeding the random number generator before its use. Next, a random number is generated and evaluated modulo 52, resulting in a random number between 0 and 51, representing a card. For a number, n, the value n / 13 is in the range 0 through 3, each corresponding to a suit: say 0 is clubs, 1 is diamonds, 2 is hearts, and 3 is spades. In addition, n % 13 + 1 evaluates to a number in the range 1 through 13, corresponding to a card in a suit: say 1 is ace, 2 is deuce, ..., 11 is jack, 12 is queen, and 13 is king.

Sample Session:

```
***Single Card Draw Program***

Type a random unsigned integer to start:   30257
Diamond 2
Heart Jack
Heart 3
Diamond Queen
Diamond 10
```

The next program uses the library function, sqrt(), to obtain square roots of randomly generated numbers. The function, sqrt(), requires its argument to be of type double, and it returns type double. In the program shown in Figure 8.5, the randomly generated whole number is assigned to a double variable before finding its square root.

Sample Session:

```
***Square Root Program - Random Numbers***

Sq.Rt.  of 346.000000 is 18.601075
Sq.Rt.  of 130.000000 is 11.401754
Sq.Rt.  of 10982.000000 is 104.795038
Sq.Rt.  of 1090.000000 is 33.015148
Sq.Rt.  of 11656.000000 is 107.962957
```

These have been just a few examples of using routines available in the math library. A complete listing of math routines is provided in Appendix C. Rather than writing our own functions all the time, we will make use of library functions in our code wherever we can in the future.

8.2 Formatted Input/Output

We have been using the I/O built-in functions printf() and scanf() which are the primary routines for formatted output and input in C (the "f" stands for formatted). We have already

```
/* File: card.c
     Program draws a card each time from a full deck of 52 cards.
*/
#include <stdio.h>
#include <math.h>

#define CLUB 0
#define DIAMOND 1
#define HEART 2
#define SPADE 3

#define ACE 1
#define JACK 11
#define QUEEN 12
#define KING 13

main()
{    int i, d1, card, suit;

     printf("***Single Card Draw Program***\n\n");
     printf("Type a random unsigned integer to start: ");
     scanf("%d", &i);
     srand(i);                     /* seed the random number generator */
     for (i = 0; i < 5; i++) {
          d1 = rand() % 52;        /* draw a card                 */
          suit = d1 / 13;          /* find the suit 0,1,2,3       */
          card = d1 % 13 + 1;      /* find the card 1, 2, ..., 13 */

          switch (suit) {          /* print suit                    */
               case CLUB: printf("Club "); break;
               case DIAMOND:  printf("Diamond "); break;
               case HEART: printf("Heart "); break;
               case SPADE: printf("Spade "); break;
          }
          switch (card) {          /* print the card within a suit */
               case ACE: printf("Ace"); break;
               case JACK: printf("Jack"); break;
               case QUEEN: printf("Queen"); break;
               case KING: printf("King"); break;
               default: printf("%d", card);
          }
          printf("\n");
     }
}
```

Figure 8.4: Program for randomly picking a card

```
/*    File sqrt3.c
      Program computes and prints square roots of numbers randomly
      generated.
*/
#include <stdio.h>
#include <math.h>

main()
{     int i;
      double x;

      printf("***Square Root Program - Random Numbers***\n\n");
      for (i = 0; i < 5; i++) {
           x = rand();
           printf("Sq.Rt. of %f is %f\n", x, sqrt(x));
      }
}
```

Figure 8.5: Code for finding the square root of random numbers

discussed many of the conversion specifications; we now present a more complete description of the formatted I/O functions together with examples. (While our discussion here concerns `printf()` and `scanf()`, it applies equally well to conversion specifications for `fprintf()` and `fscanf()` described in the next Section.

8.2.1 Formatted Output: `printf()`

As we have seen, `printf()` expects arguments giving a format string and values to be printed. The `printf()` prototype, in `stdio.h`, is:

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

The first argument of printf() is the format string (we will see what the above type declaration means in the next chapter). The number of remaining arguments depends on the number of conversion specifiers in the format string. In C, an ellipsis, i.e. "...", is used to indicate an arbitrary number of arguments. The return value of `printf()` is an `int` giving the number of bytes output, if successful; otherwise it returns `EOF`. This information from `printf()` is not generally very useful, and we often simply ignore the return value.

The function, `printf()`, converts, formats, and prints its arguments on the standard output using the conversion specifications given in the format string. The format string is made up of two kinds of characters: regular characters, which are simply copied to the output, and conversion specification characters. A conversion specification indicates how the corresponding argument value is to be converted and formatted before being printed. The number of conversion specifications in the format string must match exactly the number of arguments that follow; otherwise, the results are undefined. The data type of the argument should also match the data type it

will be converted to; for example, integral types for decimal integer formats, `float` or `double` types for floating point or exponential formats, and so on. If the proper type is not used, the conversion is performed anyway *assuming* correct data types and the results can be very strange and unexpected. Of course, character values are integral types; so characters can be converted to ASCII integer values for printing, or printed as characters. We have already seen most of the conversion characters. Table 8.1 gives a complete list with their meanings. We will discuss some examples, given the following declarations and initializations:

```
int i;
char c;
float f1;
double d1;
char *s;
long x;

i = 33;
c = 'e';
f1 = 12345.00
d1 = 12345.00
s = "This is a test";
x = 123456789;
```

Different conversion characters may be used to print the values of these variables. The space used to print a value is called the **field**, and by default, is exactly the space needed to print the value. We show examples of conversion characters and default output below:

Conversion Specifier	Variable	Output
%d	i	*33*
%o	i	*41*
%x	i	*21*
%u	i	*33*
%c	c	*e*
%d	c	*101*
%c	i	*!*
%s	s	*this is a test*
%f	f1 or d1	*12345.000000*
%e	f1 or d1	*1.234500E+004*
%g	f1 or d1	*12345*

So far, we have used very simple conversion specifiers, such as `%d`, `%f`, and `%c`. A complete conversion specification starts with the character `%` and ends with a conversion character. Between these two characters, special format characters may be used which can specify justification, field width, field separation, precision, and length modification. The characters that follow the `%` character and precede the conversion characters are called **format characters**. All format characters are optional, and if they are absent their default values are assumed. (We will indicate the default value in each case below). The syntax of a complete conversion specifier is:

Character	Conversion
d	The argument is taken to be an integer and converted to decimal integer notation.
o	The argument is taken to be an integer and converted to unsigned octal notation without a leading zero.
x	The argument is taken to be an integer and converted to unsigned hexadecimal notation without a leading `0x`.
u	The argument is taken to be an unsigned integer and converted to unsigned decimal notation.
c	The argument is taken to be an ASCII character value and converted to a character.
s	The argument is taken to be a string pointer. Unless a precision is specified as discussed below, characters from the string are printed out until a `NULL` character is reached. (Strings will be discussed further in the next chapter).
f	The argument is taken to be a `float` or `double`. It is converted to decimal notation of the form [-]ddd.dddddd, where the minus sign shown in square brackets may or may not be present. The number of digits, d, after the decimal point is 6 by default if no precision is specified. The number of digits, d, before the decimal is as required for the number.
e	The argument is taken to be a `float` or `double`. It is converted to decimal notation of the form [-]d.ddddddE[+/-]xxx, where the leading minus sign may be absent. There is one digit before the decimal point. The number of digits, d, after the decimal point is 6 if no precision is specified. The E signifies the exponent, ten, followed by a plus or minus sign, followed by the exponent. The number of digits in the exponent, x, is implementation dependent, but not less than two.
g	The same as `%e` or `%f` whichever is shorter and excludes trailing zeros.

Table 8.1: Conversion Specifier Characters for `printf()`

%-DD.dd|X

where X is one of the conversion characters from Table 8.1. The other format characters must appear in the order specified above and represent the following formatting information: (the corresponding characters are shown in parentheses).

Justification (-)   The first format character is the minus sign. If present, it specifies left justification of the converted argument in its field. The default is right justification, i.e. padding on the left with blanks if the field specified is wider than the converted argument.

Field Width (DD)   The field width is the amount of space, in character positions, used to print the data item. The digits, DD, specify the *minimum* field width. A converted argument will be printed in a field of at least this size, if it fits into it; otherwise, the field width is made large enough to fit the value. If a converted argument has fewer characters than the field width, by default it will be padded with blanks to the left, unless left justification is specified, in which case, padding is to the right.

Separator (.)   A period is used as a separator between the field width and the precision specification.

Precision (dd)   The digits, dd, specify the precision of the argument. If the argument is a `float` or `double`, this specifies the number of digits to the right of the decimal point. If an argument is a string, it specifies the maximum number of characters to be printed from the string.

Length Modifier (l)   The length modifier, l, (ell) indicates that an integer type argument is a `long` rather than an `int` type.

Some examples of format specifications using the previous variable types and values are shown below, where the field width is shown between the markers, | and |.

```
          Conversion     Variable    Output

Field Pos.                           01234567890123456789

------------------------------------------------------------
              %10d           i        |        33|
              %-10d          i        |33        |
                                      |          |
              %10f           f1       |12345.000000
              %-10f          f1       |12345.000000
              %20.2f         f1       |            12345.00|
              %-20.2f        f1       |12345.00            |
              %5c            c        |        E|
              %-5c           c        |E        |

              %10s           s        |This is a test
```

```
         %20s            s       |         This is a test|
         %-20s           s       |This is a test         |
         %20.10s         s       |          This is a    |

         %ld             x       |123456789
         %-12ld          x       |123456789    |
```

8.2.2   Formatted Input: `scanf()`

Like `printf()`, `scanf()` expects its first argument to be a format string, but unlike `printf()`, the remaining arguments are *addresses* of the variables in which to put the data that is read. The prototype for `scanf()`, also in `stdio.h`, is:

```
    int scanf(char *, ...);
```

As we've said, the returned value is the number of items read, or `EOF`. The format string controls the input order, conversion of the input data to the specified type, and format specification. Each conversion specification appearing in the format string is applied, in turn, to the next input data item in the input stream. After the specified conversion, the item is placed where the next succeeding argument points; so each of the arguments must be an address.

Besides the conversion specifications that start with `%`, the format string may also include regular characters. Regular *white space* characters in the format string are ignored. Any regular non-white space characters must be matched exactly in the input stream. For example:

```
    scanf("x= %d", &x);
```

The input stream must include the characters `x=`, which are matched by the corresponding character in the format string, before an integer value is read. A valid sample input for this format string is:

   x= *1234*

The characters, x=, are first matched, then the integer, 1234, is read and assigned to the variable, `x`. If the characters, x=, are not matched in the input stream, no input is possible, and `scanf()` will return the value 0.

As before, a conversion specification starts with a `%` and ends with one of the conversion characters given in Table 8.2. Between `%` and the conversion character, there can be an optional assignment suppression character, `*`, followed by an optional number indicating the maximum field width. The maximum field width specifies that no more than that number of characters in the input stream may be used for the next data item. The converted result is stored where the corresponding argument points unless the assignment suppression character is used. If the suppression character is used, the result is discarded. The conversion characters with their meanings are given in Table 8.2. All of these except `c` and `s` may be preceded by the length modifier, `l` (ell), where, in the case of integral type data, the corresponding argument should be `long` and in the case of floating point data, the argument should be `double`. For example, with the following declarations:

```
    int i, k;
    char c;
```

Character	Conversion
d	The input is expected to be a decimal integer. The corresponding argument should be an integer address.
o	The input is expected to be an octal number. The corresponding argument should be an integer address.
x	The input is expected to be a hexadecimal number. The corresponding argument should be an integer address.
c	The input is expected to be a character. Any character including white space may be input without being skipped over. The corresponding argument should be a character address.
s	The input is expected to be a string of characters, and the corresponding argument should be a character pointer to an array of characters large enough to accommodate the string plus the terminating `NULL` character. (Arrays are discussed in Chapter 7). The input *will* skip over initial white space and will terminate when a white space character is encountered in the input stream.
f	The input is expected to be a floating point number and the corresponding argument should be a `float` address. The input may have a sign, followed by a string of digits, optionally followed by a decimal point and a string of digits, which may be followed by an E or e and a signed or unsigned integer exponent.
e	Same as f.

Table 8.2: Conversion Specifier Characters for `scanf()`

```
float f1;
double d1;
char s[80];
long x;
```

Consider the following statements with the input as shown below each statement.

```
scanf("Integer: %4d %f", &i, &f1);
```

Input is:

Integer: 1234567

First, the regular characters, *Integer:* are matched. Then a field of 4 is used to read the integer, *1234*. Finally, a `float`, *567.0* is read. The integer, 1234, will be stored in `i`, and 567.0 will be stored in `f1`.

```
scanf("%4s %*c %c", s, &c);
```

Surprises are everywhere

A field of 4 is used to read a string, `"Surp"`, which is placed in `s`. The next character, `'r'`, will be read and discarded, and the next character, `'i'`, will be stored in `c`.

```
scanf("%s %*s %d", s, &i);
```

Surprise number 1

This time the string `"Surprise"` will be stored in `s`, the next string `"number"` will be discarded, and the integer 1 will be stored in `i`.

8.3 Direct I/O with Files

So far all our programs have used standard files for input and output; normally the keyboard and screen. Unless the standard files are redirected, users must enter data as needed, which may become inconvenient or impractical as the amount of data gets large. However, if redirection is used to read input data from other files, then *ALL* input must come from redirected files; which means the programs cannot interact with the user. Practical programs require the ability to use files for I/O as well as to interact with users via standard files. For example, data may be needed repeatedly, by different programs, over a period of time. Such data should be stored in files on disks or other peripheral devices, and programs should be able to retrieve data from these files as needed. In addition, programs can save useful data into files for later use.

In this Section, we describe some variations on our previous Input/Output routines which behave similarly, but access data directly from or to files.

8.3.1   Character I/O

We have written programs for processing characters using the routines `getchar()` and `putchar()` which read or write single characters from or to the standard input or output. The standard library provides additional, more general, routines which read or write single characters from or to any file (including `stdin` or `stdout`). We will illustrate the use of these routines with two short examples.

Our next task is to read text input from a non-standard input file and compute the frequency of occurrence of each digit in the text:

FREQ: Read input from a specified text file and calculate the frequency of occurrence of each digit in the file.

Our task calls for us to read textual data from an input file. In order for the program to be able to read from a file, the file must be identified to the program. This process is called *opening* the file. Likewise, when our use of the data in a file is complete, the file should be *closed*. Opening a file informs the program where data is to be read from, and initializes a system data structure which keeps track of how far reading has progressed in the file (along with other information needed by the operating system). Most files in C programs are treated as a *stream* of characters by the library routines that access them, and so, an open file is sometimes also referred to as a stream. Closing a file relinquishes all use of the file from the program back to the operating system. When a file is opened, the input starts at the beginning of the file and continues until the end of file is reached. The standard files, `stdin` and `stdout`, behave the same way, but they are opened automatically at the beginning of the program. They cannot be re-opened and should not normally be closed.

We can now write an algorithm for our task of counting frequency of occurrence of digits in a file (or stream). We will use an array, `digit_freq[]` to store the frequency of each digit. For each character, `ch` read, if `ch` is a digit symbol, then `ch - '0'` is the numeric equivalent of that digit and we will use `digit_freq[ch - '0']` to store the frequency of the digit. That is, `digit_freq[0]` will store frequency of digit character `'0'`, `digit_freq[1]` will store frequency for `'1'`, and so on. Here is the algorithm:

```
initialize array digit_freq[] to zero
open input file
while NOT EOF, read a character from input file stream
    if a character ch is a digit
        increment digit_freq[ch - '0']
print results to standard output
close input file
```

We begin by initializing the array, `digit_freq[]` to zero and each time a digit character is encountered, an appropriate frequency is incremented. The program implementation is shown in Figure 8.6 and assumes that the file to be read is named `test.doc`.

The input file, `test.doc` consists of a single line shown below:

245 87 129 45 28

Sample Session:

    Digit Occurrence Counter

```
/* File: cntdigits.c
    This program reads characters from a file stream and counts the
    number of occurrences of each digit.
*/
#define MAX 10
#include <stdio.h>
#include <ctype.h>       /* for isdigit() */
main()
{
    int digit_freq[MAX],i;
    signed char ch;
    FILE * fin;

    printf("***Digit Occurrence Counter***\n\n");
    /* initialize the array */
    for (i = 0; i < MAX; i++)
        digit_freq[i] = 0;

    fin = fopen("test.doc", "r");       /* open input file */
    if (!fin) {                         /* if fin is a NULL pointer */
        printf("Unable to open input file: test.doc\n");
        exit(0);                        /* exit program */
    }

    while ((ch = getc(fin)) != EOF) {  /* read a character into ch */
        if (isdigit(ch))                /* if ch is a digit */
            digit_freq[ch - '0']++;  /* increment count for digit ch */
    }
    fclose(fin);

    /* summarize */
    for (i = 0; i < MAX; i++)
        printf("There are %d occurrences of %d in the input\n",
                digit_freq[i],i);
}
```

Figure 8.6: Code for Counting Digits

```
There are 0 occurrences of 0 in the input
There are 1 occurrences of 1 in the input
There are 3 occurrences of 2 in the input
There are 0 occurrences of 3 in the input
There are 2 occurrences of 4 in the input
There are 2 occurrences of 5 in the input
There are 0 occurrences of 6 in the input
There are 1 occurrences of 7 in the input
There are 2 occurrences of 8 in the input
There are 1 occurrences of 9 in the input
```

Let us first give a summary explanation. In the declaration section of the function, `main`, a *file pointer* variable, `fin`, is declared to be of type `FILE *`. The type `FILE` is defined using a `typedef` in `<stdio.h>` as a special data structure containing the information about a file need to access it. After the array, `digit_freq[]`, is initialized to zero, the file, `test.doc`, is opened using the standard library function, `fopen()`:

```
fin = fopen("test.doc", "r");
```

The function, `fopen()`, takes two arguments: a string which gives the name of the physical file, and a second string which specifies the mode (`"r"` (for read) indicates an input file). If the file can be opened, `fopen()` returns a file pointer which can be used to access the corresponding stream. If the file cannot be opened, `fopen()` returns a `NULL` pointer, so the program tests if the returned value of the file pointer, `fin`, is `NULL` and, if so, terminates the program after a message is printed. If the file opened (i.e. `fin` is not `NULL`), then `fin` can be thought of as a "handle" on the file which is passed to an appropriate I/O routine to access the data. In our case, a character is read from the stream using the standard library function, `getc()`:

```
ch = getc(fin)
```

The function, `getc()`, reads a character from the stream accessed by the file pointer, `fin`. It returns the value of character read if successful, and `EOF` otherwise. In the program, each character read is examined to see if it is a digit; if it is, the count for that digit is incremented. Once the end of input file is reached, the file is closed with the statement:

```
fclose(fin);
```

Finally, the program prints the results accumulated in the array.

Let us now examine some details. When a file is opened, it is associated with a file buffer that serves as the interface between the physical file and the program. A program reads or writes a stream of characters from or to a file buffer. A file stream (buffer) pointer must be maintained to mark the next position in the file buffer. This information is stored in the data structure, of type `FILE`, pointed to by the file pointer. Once a physical file is opened, i.e. associated with a file buffer, and a file pointer is initialized, a program uses only the file pointer.

The derived data type, `FILE`, is defined in `<stdio.h>` using a typedef statement, and contains information about a file, such as the location of a file buffer, the current position in the buffer, file mode (read, write, append), whether errors have occurred, and whether an end of file has occurred. Users need not know the details of this data structure, instead, it is used to define pointer variables to a FILE type data item to be accessed by the library functions. For example,

```
FILE *fin, *fout;
```

declares two file pointer variables, `fin` and `fout`. It is now possible to associate these FILE pointers with desired physical files. We use the terms stream and file pointer interchangeably with FILE pointer. Standard files are always open and standard file pointer variables are available to all programs. They are named `stdin`, `stdout`, and `stderr`.

The process of opening a file connects a physical file and associates a mode with the FILE pointer. The mode specifies whether a file is opened for input, for output, or for both. The file open function, `fopen()`, associates a physical file with a file buffer or stream and returns a FILE pointer that is used to access the file. Here is the prototype:

```
FILE * fopen(char * fname, char * mode);
```

The mode string, `"r"`, specifies that the file is to be opened for reading (i.e. an input file), `"w"` specifies writing mode (i.e. an output file), and `"a"` specifies append mode (i.e. both an input and an output file). If the file was opened successfully, `fopen()` returns a pointer that will access the file stream. If it was not possible to open the file for some reason, `fopen()` returns a NULL pointer (a pointer whose value is zero — in C, the zero address is guaranteed to be an invalid address). It is the programmer's responsibility to check to see if the returned pointer is NULL. The most common reason why a file cannot be opened for reading is that it does not exist, i.e. an erroneous file name has been used.

Once a file is opened, the library function, `getc()`, reads single characters from the file stream. The argument passed to `getc()` must be a file pointer, and it returns the (integer) value of a character read or EOF if an end of file is reached.

Files should be closed after their use is completed. Failure to close open files may destroy files if a program terminates prematurely. The library function that closes a file is `fclose()`, whose argument must be a FILE pointer. The process of closing a file frees the file buffer.

In the above program, we specified the name of the input file in the code itself. If the program is to be used with any other input file, we would have to modify the program and recompile. Instead, a flexible program should ask the user to enter file names as needed.

Our next task is to copy one file to another. The algorithm is: simple.

```
get input and output file names
open files for input and output
while NOT EOF, read a character ch from input stream
    write ch to output stream
close files
```

The library routine, `putc(ch, output)` writes a character, `ch`, to a file stream, `output`. The program is shown in Figure 8.7.

Sample Session:

```
***File Copy Program - Character I/O***

Input file :  ccopy.c
Output file :  xyz.c
File copy completed
```

```
/*  File: ccopy.c
    This program copies an input file to an output file one
    character at a time. Standard files are not allowed.
*/
#include <stdio.h>

main()
{   FILE *input, *output;
    char infile[15], outfile[15];
    signed char ch;

    printf("***File Copy Program - Character I/O***\n\n");
    printf("Input file : ");
    scanf("%s", infile);
    printf("Output file : ");
    scanf("%s", outfile);

    input = fopen(infile, "r");
    if (input == NULL) {
        printf("*** Can't open input file ***\n");
        exit(0);
    }

    output = fopen(outfile, "w");
    if (output == NULL) {
        printf("*** Can't open output file ***\n");
        exit(0);
    }

    while ((ch = getc(input)) != EOF)
        putc(ch, output);
    fclose(input);
    fclose(output);
    printf("File copy completed\n");
}
```

Figure 8.7: Code to copy one file to another

The program first reads the input and output file names. We use `scanf()` to read the file names into strings, `infile` and `outfile`. These files are then opened for input and output, respectively. If either of the files cannot be opened, an error message is printed and the program is terminated by an `exit()` call. If both files are opened successfully, the copying is done in a loop until `EOF`. The loop reads a character from input into `ch` which is then written to the stream indicated by `outfile` using `putc()`. When `EOF` is reached, the files are closed and a message printed.

The file routines, `getc()` and `putc()` can be used with standard files as well. We just use the predefined file pointers for the standard files:

```
ch = getc(stdin);
putc(ch, stdout);
```

The above programs terminate if an attempt to open a file is unsuccessful. As an improvement to these programs, friendly programs should allow the user to rectify possible errors in entering file names.

8.3.2 Formatted I/O

When we read or write numeric data from or to standard file streams, `scanf()` and `printf()` convert character input to internal numeric values and vice versa. Similar functions are available for non-standard files. The function, `fscanf()` reads formatted input from a file and `fprintf()` writes formatted output to a file. The only difference between `scanf()`, `printf()` and `fscanf()`, `fprintf()` is that the latter require an additional argument which specifies the input file stream. For example, to read and write an integer from and to a file stream, we use:

```
fscanf(inp, "%d", &n);
fprintf(outp, "%d", n);
```

where `inp` and `outp`, are `FILE` pointers. The other arguments are the same as those for `scanf()` and `printf()`; the format string gives the conversion specifications, and the arguments that follow reference the objects where data is to be stored or whose values are to be written. The return value of `fscanf()` is the same as `scanf()`: number of items read or `EOF`.

Our next task is to read exam scores into an array from a file and determine the average, the maximum, and the minimum. It is assumed that the data file of exam scores is prepared using an editor. The algorithm is simple enough:

```
get input file name
open input file
read exam scores into an array
process the array to find average, maximum, and minimum
```

We will use a function, `proc_aray()`, to process the array. It will return the average but will indirectly store the maximum and minimum values in the calling function. The program is shown in Figure 8.8.

The sample session assumes that the scores are in an input file `scores.dat` prepared using an editor and shown below:

```c
/*  File: avgfile.c
    This program reads exam scores from a file and processes them to
    find the average, the maximum, and the minimum.  */
#include <stdio.h>
#define MAX 100
float proc_aray(int ex[], int lim, int *pmax, int *pmin);
main()
{   int max, min, n, lim = 0, exam_scores[MAX];
    char infile[15];
    float avg;
    FILE * inp;

    printf("***Exam Scores: Average, Maximum, Minimum***\n\n");
    printf("Input File: ");
    scanf("%s", infile);
    inp = fopen(infile, "r");
    if (!inp) {
        printf("Unable to open input file\n");
        exit(0);
    }
    while (lim < MAX && fscanf(inp, "%d", &n) != EOF)
        exam_scores[lim++] = n;
    fclose(inp);
    if (lim == 0) exit(0);
    avg = proc_aray(exam_scores, lim, &max, &min);
    printf("Average = %f, Maximum = %d, Minimum = %d\n",
            avg, max, min);
}


/*  This function computes the average of an array, the maximum and
    the minimum. Average is returned, the others are indirectly
    stored in the calling function.  */
float proc_aray(int ex[], int lim, int *pmax, int *pmin)
{   int i, max, min;
    float sum = 0.0;

    max = min = ex[0];
    for (i = 0; i < lim; i++) {
        sum += ex[i];
        max = ex[i] > max ? ex[i] : max;
        min = ex[i] < min ? ex[i] : min;
    }
    *pmax = max;
    *pmin = min;
    return sum / lim;
}
```

Figure 8.8: Code for avgfile.c

```
67
75
82
69
```

Sample Session:

```
***Exam Scores:  Average, Maximum, Minimum***

Input File:  scores.dat
Average = 73.250000, Maximum = 82, Minimum = 67
```

The driver opens the input file and reads data into the array, `exam_scores[]`. The number of elements are counted by `lim`. If `lim` is zero, the program is terminated; otherwise, the program calls `proc_aray()` to process the array for the average, the maximum, and the minimum. In the call to `proc_aray()`, `main()` passes as arguments `exam_scores`, `lim`, and pointers to `max` and `min`.

The function, `proc_aray()`, initializes values of local variables, `max` and `min`, to the value of the first element of the array, `ex[0]`. It then traverses the array, maintains a cumulative sum of the scores, and updates the values of `max` and `min` using the following conditional expressions:

```
max = ex[i] > max ? ex[i] : max;
min = ex[i] < min ? ex[i] : min;
```

Here, if an array element, `ex[i]`, is greater than `max`, `max` is assigned `ex[i]`; otherwise, `max` is assigned `max`. Similarly, the minimum is updated when an array element is smaller than the minimum. Finally, the function indirectly stores values of maximum and minimum, and returns the value of the average score.

8.4 Common Errors

1. Use of `scanf()` to read strings with white space. When `scanf()` is used to read a string, only part of an input string may be read: it skips over leading white space, and reads a string until the next white space.

   ```
   scanf("%s", msg);
   ```

   Input: `this is a string`

   With the above input, `scanf()` will read "this", and NOT the whole string, into memory pointed to by `msg`. However, `printf()` will print the entire string until the terminating NULL.

8.5 Summary

In this Chapter we have discussed various features available to the programmer in the C standard library. While we have used some of the functions in previous chapters, particularly those for I/O, we have given a more detailed description of the library, and the standard I/O routines

provided there. We have seen that frequently used operations on characters for classifying or converting which we have written for ourselves in the past, are available from the library. In addition, routines for common math operations are also provided in the math library (which may not be automatically linked by the compiler). We have given a few short programs illustrating the use of some of these functions. A more complete list of available library routines is provided in Appendix C.

We have also given a complete description of the formatted I/O functions, `scanf()` and `printf()` detailing the options available for formatting input and output. Finally, we have discussed variations on the I/O routines available, both for characters and formatted, which allow direct access to data in files from within a program. These new routines include `getc()`, `putc()`, `fscanf()`, and `fprintf()`, as well as functions for managing connection to the physical files: `fopen()` and `fclose()`.

The full power of the C standard library is now available for future program development in later chapters.

8.6   Exercises

1.
```
main()
{    long n;

     scanf("%d", &n);
}
```

2.
```
main()
{    long n = 12L;

     printf("%d\n", n);
}
```

3.
```
main()
{    double x;

     scanf("%f", &x);
}
```

4. If **x** is 100 and **z** is 200, what is the output of the following:

```
if (z = x)
     printf("z = %d, x = %d\n", z, x);
```

8.7   Problems

1. Write a program to make a table of decimal, octal, and hexadecimal unsigned integers from 0 to 255.

2. Write a program to print a calendar for a month, given the number of days in the month and the day of the week for the first day of the month. For example, given that there are 30 days and the first of the month is on Tuesday, the program should print the calendar for the month.

```
        CALENDAR FOR THE MONTH
   sun  mon  tue  wed  thu  fri  sat
              1    2    3    4    5
   6    7    8    9    10   11   12
   13   14   15   16   17   18   19
   20   21   22   23   24   25   26
   27   28   29   30
```

3. Write a program to read the current date in the order: year, month, and day of month. The program then prints the date in words: Today is the nth day of Month of the year Year. Example:

```
Today is the 24th day of December of the year 2000.
```

4. Write a program that prints a calendar for a year given the day of the week on the first day of the year. (Use Problem 3.6 for the definition of a leap year).

5. Write a program that prints a calendar for any year in this century given the day of the week for the first day of the current year.

6. Write a function that returns the value of a random throw of two separate dice.

7. Write the following functions:

```
first\_card() that draws a random card from a full deck.
second\_card() that draws a random card from the remaining deck.
Similarly, write third\_card() and fourth\_card().
```

For the last three functions, you will need arguments that indicate what cards have already been drawn from the deck.

8. Write a program using the functions of Problem ? to play a game of "black jack" with the user. Each side is dealt cards alternately. First each side is dealt two cards, but one at a time. Then, if necessary a maximum of one more card is allowed for each player. The player with the highest score, not exceeding 21, wins. In a tie, the user wins. The program should reshuffle the cards and play the game as long as the user wishes. The score is obtained by summing the value of each card. The value of a card is the face value of the card, except that an ace can be either 1 or 11 and all picture cards are 10.

9. Randomly toss a coin: repeat and count the number of heads and tails in 100 tosses, 500 tosses, 1000 tosses.

10. Write a program to play a board game with the user. The game uses a throw of two dice. The rules of the game are as follows. Each player takes a turn and is allowed a succession of throws. If a player's first throw is seven or eleven, he/she loses the turn. Otherwise, the player's score is increased by the value of each throw until the dice show a seven or a eleven. The turns continue between the user and the program until a pre-set limit for the score is reached.

11. Write a program to compare the routine sq_root() written in Chapter ? with the standard library routine. How close are the routines?

12. Write a function that returns the hypotenuse, given the two sides of a right angled triangle. A hypotenuse is the square root of the sum of the squares of the two sides of a right angled triangle.

13. Find all the angles of a right angled triangle if the lengths of the two sides are given. Since it is a right triangle, one angle is pi / 2. Also, the ratio of the two sides in a right triangle gives the tangent of one of the other angles. Therefore, one angle is the arctangent of the ratio of the two sides. Use a library function to get the arctangent of a value. The other angle is easily obtained since the three angles must add up to pi.

14. Use library routines to compare values of sine, cosine, and exponential with those calculated by Chapter 3 problems 3.1 through 3.3.

15. Write a menu-driven program that handles the grades for a class. The program allows the following commands.

    ```
    Get data: gets id numbers and integer scores for a set of 3 projects from a
file. Assume that the id numbers start at 0 and go up to a maximum of 99.
    Print data: prints the scores.
    Average scores: averages each set of scores.
    Change scores: allows the user to make changes in scores for any project
and for any id number.
    ```

16. Write a program that reads a text of characters from a file and keeps track of the frequency of usage of each letter, digit, and punctuation.

17. Write a menu-driven program that reads input data from a file. The program reads and stores for each student the ID number, course numbers, credits, and grades. Assume a maximum of 3 courses per student. The program should compute and store the GPR for each student. At the end of input, the program writes to a file as well as to the standard output all the input data and GPR for all students.

18. Write a program that shuffles and deals out all 52 cards of a deck of playing cards to 4 players. Each card is dealt in sequence around a table to players in the following order: west, north, east, south. Print out the hands of each player. You must use a random generator, but discard a possible card if it has already been dealt out. Use an array of 52 elements to keep track of the cards already dealt out.

19. Write a program to play the game of 21 with a limit of five cards for each player. Assume the program plays south and deals the cards. The other three players are in order west, north, and east. Cards must be dealt randomly.

20. Write a program that reads a positive integer n; it then generates a set of n random numbers in a range from 0 to 99. Store and count the frequency of occurrence of each number. Print the frequency of each number.

21. Use an array to read from a file and store the sample values of an experiment at regular intervals. Plot the graph of the sample values versus time. Time should increase vertically downwards. Use '*' to mark a point. Write a program to read in sample values and call a function to plot the values.

22. Repeat Problem 27, but plot a bar chart for the sample values.

Chapter 9

Two Dimensional Arrays

In Chapter 7 we have seen that C provides a compound data structure for storing a list of related data. For some applications, however, such a structure may not be sufficient to capture the organization of the data. For example, in our payroll task, we have several pieces of information (hours worked, rate of pay, and regular and over time pay) for a list of employees. We have organized such data as individual arrays, one for each "column", to form the payroll data base; but conceptually, this information is all related. In this chapter we will introduce a data structure which allows us to group together all such information into a single structure — a two dimensional array. For a data base application, we can think of this 2D organization as an array of arrays. As another example of where such a structure is convenient, consider an array of names. We have seen that we can store a name in a *string*, which is an array of characters. Then an array of strings is also an array of arrays, or a two dimensional array.

In this chapter we will discuss how we can declare two dimensional arrays, and use them in applications. We will see how we can access the data in such a structure using indices and pointers, and see how this concept can be extended to multi-dimensional arrays. We will present examples of 2 dimensional arrays for data base applications, string sorting and searching, and solutions to systems of simultaneous linear algebraic equations, useful in scientific, engineering, and other applications, e.g. electronic circuit analysis, economic analysis, structural analysis, etc. The one restriction in the use of this data type is that all of the data stored in the structure must be of the same type. (We will see how we can remove this restriction in the next chapter).

9.1 Two Dimensional Arrays

Our first task is to consider a number of exams for a class of students. The score for each exam is to be weighted differently to compute the final score and grade. For example, the first exam may contribute 30% of the final score, the second may contribute 30%, and the third contribute 40%. We must compute a weighted average of the scores for each student. The sum of the weights for all the exams must add up to 1, i.e. 100%. Here is our task:

WTDAVG: Read the exam scores from a file for several exams for a class of students. Read the percent weight for each of the exams. Compute the weighted average score for each student. Also, compute the averages of the scores for each exam and for the weighted average scores.

We can think of the exam scores and the weighted average score for a single student as a data record and and represent it as a row of information. The data records for a number of students,

then, is a table of such rows. Here is our conceptual view of this collection of data:

	exam1	exam2	exam3	weighted avg
$student_1$	50	70	75	??
$student_2$	90	95	100	??
$student_3$	89	87	92	??
$\vdots$				
$student_n$	90	90	91	??

Let us assume that all scores will be stored as integers; even the weighted averages, which will be computed as float, will be rounded off and stored as integers. To store this information in a data structure, we can store each student's data record, a row containing three exam scores and the weighted average score, in a one dimensional array of integers. The entire table, then, is an array of these one dimensional arrays — i.e. a two dimensional array. With this data structure, we can access a record for an individual student by accessing the corresponding row. We can also access the score for one of the exams or for the weighted average for all students by accessing each column. The only restriction to using this data structure is that all items in an array must be of the same data type. If the student id is an integer, we can even include a column for the id numbers.

Suppose we need to represent id numbers, scores in 3 exams, and weighted average of scores for 10 students; we need an array of ten data records, one for each student. Each data record must be an array of five elements, one for each exam score, one for the weighted average score, and one for the student id number. Then, we need an array, `scores[10]` that has ten elements; each element of this array is, itself, an array of 5 integer elements. Here is the declaration of an array of integer arrays:

```
int scores[10][5];
```

The first range says the array has ten elements: `scores[0]`, `scores[1]`, ... `scores[9]`. The second range says that each of these ten arrays is an array of five elements. For example, `scores[0]` has five elements: `scores[0][0]`, `scores[0][1]`, ... `scores[0][4]`. Similarly, any other element may be referenced by specifying two appropriate indices, `scores[i][j]`. The first array index references the ith one dimensional array, `scores[i]`; the second array index references the jth element in the one dimensional array, `scores[i][j]`.

A two dimensional array lends itself to a visual display in rows and columns. The first index represents a row, and the second index represents a column. A visual display of the array, `scores[10][5]`, is shown in Figure 9.1. There are ten rows, (0-9), and five columns (0-4). An element is accessed by row and column index. For example, `scores[2][3]` references an integer element at row index 2 and column index 3.

We will see in the next section that, as with one dimensional arrays, elements of a two dimensional array may be accessed indirectly using pointers. There, we will see the connection between two dimensional arrays and pointers. For now, we will use array indexing as described above and remember that arrays are always accessed indirectly. Also, just as with one dimensional arrays, a 2D array name can be used in function calls, and the called function accesses the array indirectly.

We can now easily set down the algorithm for our task:

	col. 0	col. 1	col. 2	col. 3	col. 4
row 0	scores[0][0]	scores[0][1]	scores[0][2]	scores[0][3]	scores[0][4]
row 1	scores[1][0]	scores[1][1]	scores[1][2]	scores[1][3]	scores[1][4]
row 2	scores[2][0]	scores[2][1]	scores[2][2]	scores[2][3]	scores[2][4]
row 3	scores[3][0]	scores[3][1]	scores[3][2]	scores[3][3]	scores[3][4]
row 4	scores[4][0]	scores[4][1]	scores[4][2]	scores[4][3]	scores[4][4]
row 5	scores[5][0]	scores[5][1]	scores[5][2]	scores[5][3]	scores[5][4]
row 6	scores[6][0]	scores[6][1]	scores[6][2]	scores[6][3]	scores[6][4]
row 7	scores[7][0]	scores[7][1]	scores[7][2]	scores[7][3]	scores[7][4]
row 8	scores[8][0]	scores[8][1]	scores[8][2]	scores[8][3]	scores[8][4]
row 9	scores[9][0]	scores[9][1]	scores[9][2]	scores[9][3]	scores[9][4]

Figure 9.1: Rows and Columns in A Two Dimensional Array

```
read the number of exams into no_of_exams
get weights for each of the exams

read exam scores and id number for each student
     into a two dimensional array

for each student, compute weighted average of scores in the exams
compute average score for each of the exams and
     for the weighted average
print results
```

We can easily write the top level program driver using functions to do the work of reading scores, getting the weights, computing the weighted averages, printing scores, averaging each set of scores, and printing the averages. The driver is shown in Figure 9.2.

We have declared an array, `scores[][]`, with `MAX` rows and `COLS` columns, where these macro values are large enough to accommodate the expected data. We have used several functions, which we will soon write and include in the same program file. Their prototypes as well as those of other functions are declared at the head of the file. In the driver, `getwts()` reads the weights for the exams into an array, `wts[]`, returning the number of exams. The function, `read_scores()`, reads the data records into the two dimensional array, `scores[][]`, and returns the number of data records. The function, `wtd_avg()`, computes the weighted averages of all exam scores, and `avg_scores()` computes an average of each exam score column as well as that of the weighted average column. Finally, `print_scores()` and `print_avgs()` print the results including the input data, the weighted averages, and the averages of the exams.

Let us first write `getwts()`. It merely reads the weight for each of the exams as shown in Figure 9.3. The function prompts the user for the number of exam scores, and reads the corresponding number of float values into the `wts[]` array. Notice that the loop index, `i` begins with the value 1. This is because the element `wts[0]`, corresponding to the student id column, does not have a weight and should be ignored. After the weights have been read, we flush the keyboard buffer of any remaining white space so that any kind of data (including character data) can be read from

```
/*    File: wtdavg.c
      Other Source Files: avg.c
      Header Files: avg.h
      This program computes weighted averages for a set of exam scores for
      several individuals. The program reads scores from a file, computes
      weighted averages for each individual, prints out a table of scores,
      and prints averages for each of the exams and for the weighted average.
*/

#include <stdio.h>

#define MAX 20
#define COLS 5
int getwts(float wts[]);
FILE *openinfile(void);
int read_scores(int ex[][COLS], int lim, int nexs);
void wtd_avg(int ex[][COLS], int lim, int nexs, float wts[]);
void avg_scores(int ex[][COLS], int avg[], int lim, int nexs);
void print_scores(int ex[][COLS], int lim, int nexs);
void print_avgs(int avg[], int nexs);

main()
{     int no_of_stds, no_of_exams;
      int avg[COLS];
      int scores[MAX][COLS];
      float wts[COLS];

      printf("***Weighted Average of Scores***\n\n");
      no_of_exams = getwts(wts);
      no_of_stds = read_scores(scores, MAX, no_of_exams);

      wtd_avg(scores, no_of_stds, no_of_exams, wts);
      print_scores(scores, no_of_stds, no_of_exams);
      avg_scores(scores, avg, no_of_stds, no_of_exams);
      print_avgs(avg, no_of_exams);
}
```

Figure 9.2: Driver for Student Scores Program

```
/* File: wtdavg.c - continued */
/* Gets the number of exams and weights for the exams; flushes
   the input buffer and returns the number of exams.
*/
int getwts(float wts[])
{    int i, n;

     printf("Number of exams: ");
     scanf("%d", &n);

     for (i = 1; i <= n; i++) {
     printf("Percent Weight for Exam %d: ", i);
     scanf("%f", &wts[i]);
     }

     while (getchar() != '\n')
          ;
     return n;
}
```

Figure 9.3: Code for `getwts()`

the input. The function returns the number of exams, `n`.

We will assume that the data for the student scores is stored in a file in the format of one line per student, with each line containing the student id followed by the exam scores. To read this data into a two dimensional array, we must first open the input file. This is done by the function `openfile()` shown in Figure 9.4, which prompts for the file name and tries to open the file. If the file opens successfully, the file pointer is returned. Otherwise, the function prints a message and asks the user to retype the file name. The user may quit at any time by typing a newline or end of file. If an end of file is typed or the typed string is empty, the program is terminated. Once the input file is opened, we read data items into the array, filling in the elements one row (student) at a time. We use two index variables, `row` and `col`, varying the `row` to access each row in turn; and, within each row, we vary `col` to access elements of each column in turn. We will need a doubly nested loop to read the data in this manner. The function is given the number of students, the variable `stds`, and the number of exams, `nexs`. We will use column 0 to store the student id numbers and the next `nexs` columns to store the scores. Thus, in each row, we read `nexs+1` data values into the array. This is done by the function, `read_scores()`, also shown in Figure 9.4. The input file is first opened using `openfile()`, and the data records are read into the array called `ex[][]` within the function. The function returns the number of records read either when `EOF` is reached or when the array is filled. Each integer data item is read from a file, `fp`, into a temporary variable, `n`. This value is then assigned to the appropriate element, `ex[row][col]`. When all data has been read, the input file is closed and the number of records read is returned.

Notice in `main()` in Figure 9.2, we pass the 2D array to `read_scores()` just as we did for one dimensional arrays, passing the array name. As we shall see in the next section, the array

```
/* File: wtdavg.c - continued */
/* Opens the input file and returns the file pointer. */
FILE *openinfile(void)
{    FILE *fp;
     char infile[25];

     printf("Input File, RETURN to quit: ");
     while (gets(infile)) {
          if (!*infile) exit(0);    /* empty string, exit */

          fp = fopen(infile, "r");

          if (!fp) {               /* no such file, continue */
               printf("Unable to open input file - retype\n");
               continue;
          }

          else return fp;       /* file opened, return fp */
     }
     exit(0);                     /* end of file typed, exit */
}


/* Opens the input file and reads scores for nexs exams; returns
   the number of individual student records.
*/
int read_scores(int ex[][COLS], int stds, int nexs)
{    int row, col, n, x;
     FILE * fp;

     fp = openinfile();
     for (row = 0; row < stds; row++)
          for (col = 0; col <= nexs; col++) {
               x = fscanf(fp, "%d", &n);

               if (x == EOF) {
                    fclose(fp);
                    return row;
               }

               ex[row][col] = n;
          }
     fclose(fp);
     return row;
}
```

Figure 9.4: Code for openfile() and read_scores()

name is a pointer that allows indirect access to the array elements. The two dimensional array as as argument must be declared in the function definition as a formal parameter. In Figure 9.4, we have declared it as `ex[][COL]` with two sets of square brackets to indicate that it points to a two dimensional array. In our declaration, we **must** include the number of columns in the array because this specifies the size of each row. Recall, the two dimensional array is an array of rows. Once the compiler knows the size of a row in the array, it is able to correctly determine the beginning of each row.

The next function called in `main()` computes the weighted average for each row. The weighted average for one record is just the sum of each of the exam score times the actual weight of that exam. If the scores are in the array, `ex[][]`, then the following code will compute a weighted average for a single row, `row`:

```
wtdavg = 0.0;
for (col = 1; col <= nexs; col++)
        wtdavg += ex[row][col] * wts[col] / 100.0;
```

We convert the percent weight to the actual weight multiply by the score, and accumulate it in the sum, `wtdavg` yielding a float value. The `wtdavg` will be stored in the integer array, `ex[][]`, after rounding to a nearest integer. If we simply cast `wtdavg` to an integer, it will be truncated. To round to the nearest integer, we add 0.5 to `wtdavg` and then cast it to integer:

```
ex[row][nexs + 1] = (int) (0.5 + wtdavg);
```

The weighted average is stored into the column of the array after the last exam score. The entire function is shown in Figure 9.5

Computing average of each of the exams and the weighted average is simple. We just sum each of the columns and divide by the number of items in the column, and is also shown in Figure 9.5. For each exam and for the weighted average column, the scores are added and divided by `lim`, the number of rows in the array, using floating point computation. The result is rounded to the nearest integer and stored in the array, `avg[]`. Figure 9.6 shows the final two functions for printing the results.

Running the program with data file,`wtdin.dat` as follows:

```
3    70    76
52    92    80
53    95    56
54    48    52
55    98    95
57    100    95
61    100    65
62    95    76
63    86    65
70    100    90
71    73    73
75    94    79
```

produces the following sample session:

```
***Weighted Average of Scores***
```

```
/* File: wtdavg.c - continued */
/* Computes the weighted average of the exam scores in ex[][] for
   lim individuals, nexs number of exams, and weights given by wts[].
*/
void wtd_avg(int ex[][COLS], int lim, int nexs, float wts[])
{   int i, j;
    float wtdavg;

    for (i = 0; i < lim; i++) {
        wtdavg = 0.0;

        for (j = 1; j <= nexs; j++)
            wtdavg += ex[i][j] * wts[j] / 100.0;

        ex[i][nexs + 1] = (int) (wtdavg + 0.5);
    }
}


/* Averages exam and weighted average scores. */
void avg_scores(int ex[][COLS], int avg[], int lim, int nexs)
{   int i, j;

    for (j = 1; j <= nexs + 1; j++) {
        avg[j] = 0;

        for (i = 0; i < lim; i++)
            avg[j] += ex[i][j];

        avg[j] = (int) (0.5 + (float) avg[j] / (float) lim);
    }
}
```

Figure 9.5: Code for wtd_avg() and avg_scores()

```
/* File: wtdavg.c - continued */
/* Prints the scores for exams and the weighted average. */
void print_scores(int ex[][COLS], int lim, int nexs)
{    int i, j;

     printf("ID #\t");
     for (j = 1; j <= nexs; j++)
          printf("Ex%d\t", j);              /* print the headings */

     printf("WtdAvg\n");
     for (i = 0; i < lim; i++) {            /* print the scores and wtd avg */

          for (j = 0; j <= nexs + 1; j++)
               printf("%d\t", ex[i][j]);

          printf("\n");
     }
}

/* Prints the averages of exams and the average of the weighted average
   of exams.
*/
void print_avgs(int avg[], int nexs)
{    int i;

     for (i = 1; i <= nexs; i++)
          printf("Average for Exam %d = %d\n", i, avg[i]);
     printf("Average of the weighted average = %d\n", avg[nexs + 1]);
}
```

Figure 9.6: Code for print_scores() and print_avgs()

```
Number of exams:   2
Percent Weight for Exam 1:   50
Percent Weight for Exam 2:   50
Input File, RETURN to quit:   wtdin.dat
 ID #   Ex1   Ex2   WtdAvg
 3      70    76    73
 52     92    80    86
 53     95    56    76
 54     48    52    50
 55     98    95    97
 57     100   95    98
 61     100   65    83
 62     95    76    86
 63     86    65    76
 70     100   90    95
 71     73    73    73
 75     94    79    87
Average for Exam 1 = 88
Average for Exam 2 = 75
Average of the weighted average = 82
```

In this program, we have assumed that the input file contains only the data to be read, i.e. the student id numbers and exam scores. Our `read_scores()` function is written with this assumption. However, the input file might also contain some heading information such as the course name and column headings in the first few lines of the file. We can easily modify `read_scores()` to discard the first few lines of headings.

As a second example of application of two dimensional arrays, consider our previous payroll example. In this case, the data items in a pay data record are not all of the same data type. The id numbers are integers, whereas all the other items are float. Therefore, we must use an array of integers to store the id numbers, and a two dimensional float array to store the rest of the data record. The algorithm is no different from the program we developed in Chapter 7 that computed pay. The difference is that now we use a two dimensional array for all float payroll data instead of several one dimensional arrays. The id numbers are still stored in a separate one dimensional array. Since the data structures are now different, we must recode the functions to perform the tasks of getting data, calculating pay, and printing results, but still using the same algorithms.

The program driver and the header files are shown in Figure 9.7. The program declares an integer array for id numbers and a two dimensional float array for the rest of the data record. The successive columns in the two dimensional array store the hours worked, rate of pay, regular pay, overtime pay, and total pay, respectively. We have defined macros for symbolic names for these index values. As in the previous version, the program gets data, calculates pay, and prints data. The difference is in the data structures used. Functions to perform the actual tasks are shown in Figure 9.8 and 9.9 and included in the same program source file. Each function uses a two dimensional array, `payrec[][]`. The row index specifies the data record for a single id, and the column index specifies a data item in the record. The data record also contains the total pay. A sample interaction with the program, `pay2rec.c`, is shown below.

```
/*    File: pay2rec.c
      Program calculates and stores payroll data for a number of id's.
      The program uses a one dimensional array for id's, and a two
      dimensional array for the rest of the pay record. The first column
      is hours, the second is rate, the third is regular pay, the fourth
      is overtime pay, and the fifth is total pay.
*/

#include <stdio.h>
#define MAX 10
#define REG_LIMIT 40
#define OT_FACTOR 1.5
#define HRS 0
#define RATE 1
#define REG 2
#define OVER 3
#define TOT 4

int get2data(int id[], float payrec[][TOT + 1], int lim);
void calc2pay(float payrec[][TOT + 1], int n);
void print2data(int id[], float payrec[][TOT + 1], int n);

main()
{     int n = 0, id[MAX];
      float payrec[MAX][TOT + 1];

      printf("***Payroll Program - Records in 2 D arrays***\n\n");
      n = get2data(id, payrec, MAX);
      calc2pay(payrec, n);
      print2data(id, payrec, n);
}
```

Figure 9.7: Driver for Payroll Program Using 2D Arrays

```
/*   File: pay2rec.c - continued */
/*   Gets id's in one array, and the rest of input data records
     in a two dimensional array.
*/
int get2data(int id[], float payrec[][TOT + 1], int lim)
{    int n = 0;
     float x;

     while (n < lim) {
          printf("ID <zero to quit>: ");
          scanf("%d", &id[n]);

          if (id[n] <= 0)
               return n;

          printf("Hours Worked: ");
          scanf("%f", &x);
          payrec[n][HRS] = x;

          printf("Rate of Pay: ");
          scanf("%f", &x);
          payrec[n][RATE] = x;
          n++;
     }
     if (n == lim) {
          printf("Table full, processing data\n");
          return n;
     }
}
```

Figure 9.8: Code for Payroll Program Functions — get2data()

```
/* Calculates pay for each id record in a two dimensional array. */
void calc2pay(float payrec[][TOT + 1], int n)
{    int i;

     for (i = 0; i < n; i++) {
          if (payrec[i][HRS] <= REG_LIMIT) {
               payrec[i][REG] = payrec[i][HRS] * payrec[i][RATE];
               payrec[i][OVER] = 0;
          }

          else {
               payrec[i][REG] = REG_LIMIT * payrec[i][RATE];
               payrec[i][OVER] = (payrec[i][HRS] - REG_LIMIT) *
                                      OT_FACTOR * payrec[i][RATE];
          }

          payrec[i][TOT] = payrec[i][REG] + payrec[i][OVER];
     }
}


/*   Prints a table of payroll data for all id's. Id's in one array,
     and the rest of the records in a two dim. array.
*/
void print2data(int id[], float payrec[][TOT + 1], int n)
{    int i, j;

     printf("***PAYROLL: FINAL REPORT***\n\n");
     printf("%10s%10s%10s%10s%10s%10s\n", "ID", "HRS",
               "RATE", "REG", "OVER", "TOT");

     for (i = 0; i < n; i++) {
          printf("%10d", id[i]);

          for (j = 0; j <= TOT; j++)
               printf("%10.2f", payrec[i][j]);

          printf("\n");
     }
}
```

Figure 9.9: Code for Payroll Program Functions — calc2pay() and print2data()

Sample Session:

```
***Payroll Program - Records in 2 D arrays***

ID <zero to quit>:   5
Hours Worked:  30
Rate of Pay:  10
ID <zero to quit>:   10
Hours Worked:  50
Rate of Pay:  12
ID <zero to quit>:   0
***PAYROLL: FINAL REPORT***
 ID   HRS   RATE     REG      OVER      TOT
  5  30.00  10.00  300.00     0.00   300.00
 10  50.00  12.00  480.00   180.00   660.00
```

9.2   Implementing Multi-Dimensional Arrays

In the last section we saw how we can use two dimensional arrays — how to declare them, pass them to functions, and access the date elements they contain using array indexing notation. As with one dimensional arrays, we can access elements in a 2D array using pointers as well. In order to understand how this is done, in this section we look at how multi dimensional arrays are implemented in C.

As we saw in Chapter 7, a one dimensional array is stored in a set of contiguous memory cells and the *name* of the array is associated with a pointer cell to this block of memory. In C, a two dimensional array is considered to be a one dimensional array of rows, which are, themselves, one dimensional arrays. Therefore, a two dimensional array of integers, AA[][], is stored as a contiguous sequence of elements, each of which is a one dimensional array. The rows are stored in sequence, starting with row 0 and continuing until the last row is stored, i.e. AA[0] is stored first, then AA[1], then AA[2], and so on to AA[MAX-1]. Each of these "elements" is an array, so is stored as a contiguous block of integer cells as seen in Figure 9.10. This storage organization for two dimensional arrays is called row major order. The same is true for higher dimensional arrays. An n dimensional array is considered to be a one dimensional array whose elements are, themselves, arrays of dimension $n - 1$. As such, in C, an array of any dimension is stored in row major order in memory.

With this storage organization in mind, let us look at what implications this has to referencing the array with pointers. Recall that an array name (without an index) represents a pointer to the first object of the array. So the name, AA, is a pointer to the element AA[0]. iBut, AA[0] is a one dimensional array; so, AA[0] points to the first object in row 0, i.e. AA[0] points to AA[0][0]. Similarly, for any k AA[k] points to the beginning of the kth row, i.e. AA[k] is the address of AA[k][0]. Since AA[k] points to AA[k][0], *AA[k] accesses AA[k][0], an object of type integer. If we add 1 to the pointer AA[k], the resulting pointer will point to the next integer type element, i.e. the value of AA[k][1]. In general, AA[k] + j points to AA[k][j], and *(AA[k] + j) accesses the value of AA[k][j]. This is shown in Tables 9.1 and 9.2. Each AA[k] points to an integer type object. When an integer is added to the pointer AA[k], the resulting pointer points to the next object of the integer type.
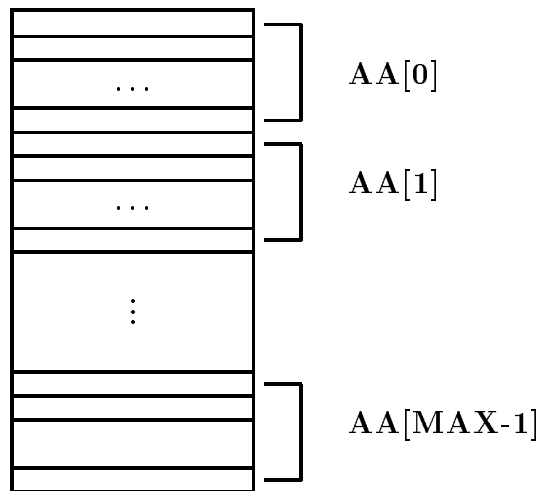
Figure 9.10: A Two Dimensional Array in Row Major Order

AA[0]	&AA[0][0]
AA[1]	&AA[1][0]
AA[2]	&AA[2][0]
AA[k]	&AA[k][0]
AA[0] + 1	&AA[0][1]
AA[0] + j	&AA[0][j]
AA[k] + j	&AA[k][j]

Table 9.1: Array Pointers and Sub-Arrays

* AA[0]	AA[0][0]
AA[k]	AA[k][0]
(AA[0] + 1)	AA[0][1]
(AA[0] + j)	AA[0][j]
(AA[k] + j)	AA[k][j]

Table 9.2: Dereferencing Array Pointers

* AA	AA[0]	&AA[0][0]
AA + 1	AA[0] + 1	&AA[0][1]
AA + j	AA[0] + j	&AA[0][j]
(AA + 1)	AA[1]	&AA[1][0]
(AA + k)	AA[k]	&AA[k][0]
(* AA)	*AA[0]	AA[0][0]
(* (AA + 1))	*AA[1]	AA[1][0]
(* (AA + k) + j)	*(AA[k] + j)	AA[k][j]

Table 9.3: Pointer Equivalence for Two Dimensional Arrays

The name, `AA`, is the name of the entire array, whose elements are themselves arrays of integers. Therefore, `AA` points to the first object in this array of arrays, i.e. `AA` points to the array `AA[0]`. The addresses represented by `AA` and `AA[0]` are the same; however, they point to objects of different types. `AA[0]` points to `AA[0][0]`, so it is an integer pointer. `AA` points to `AA[0]`, so it is a pointer to an integer pointer. If we add 1 to `AA`, the resulting pointer, `AA + 1`, points to the array `AA[1]`, and `AA + k` points to the array `AA[k]`. When we add to a pointer to some type, we point to the next object of that type. Therefore, adding to `AA` and `AA[0]` result in pointers to different objects. Adding 1 to `AA` results in a pointer that points to the next array or row, i.e. `AA[1]`; whereas adding 1 to `AA[0]` results in a pointer that points to `AA[0][1]`. Dereferencing such a pointer, `*(AA + k)`, accesses `AA[k]`, which as we saw, was `&AA[k][0]`. It follows that `*(*(AA + k) + j)` accesses the integer, `AA[k][j]`. This pointer equivalence for two dimensional arrays is shown in Table 9.3.

The C compiler converts array indexing to indirect access by dereferenced pointers as shown in the table; thus, all array access is indirect access When we pass a 2D array to a function, we pass its name (`AA`). The function can access elements of the array argument either by indexing or by pointers. We generally think of a two dimensional array as a table consisting of rows and columns as seen in Figure 9.11. As such, it is usually easiest to access the elements by indexing; however, the pointer references are equally valid as seen in the figure.

The relationships between different pointers for a two dimensional array is further illustrated with the program shown in Figure 9.12. The two-dimensional array, `a`, is not an integer pointer, it points to the array of integers, `a[0]`. However, `*a` is an integer pointer; it points to an integer object, `a[0][0]`. To emphasize this point, we initialize an integer pointer, `intptr` to `*a`, i.e. `a[0]`. The initial value of `intptr` is the address of `a[0][0]`. We next print the values of `a` and `*a`, which are the same address even though they point to different types of objects. In the `for` loop, we print the value of `a + i`, which is the same as that of `a[i]` even though they point to different types of objects. In the inner for loop, we print the address of the ith row and the jth column element of the row major array using pointers:

```
*a + COL * i + j
```

The same value is printed using the *address of* operator, `&a[i][j]`. Finally, the value of `a[i][j]` is printed using array indices as well as by dereferencing pointers, i.e. `*(*(a + i) + j)`.

The value of `intptr`, initialized to `*a`, is incremented after each element value is printed;

AA[0]+1 AA[0]+2   AA[0]+MAXCOL-
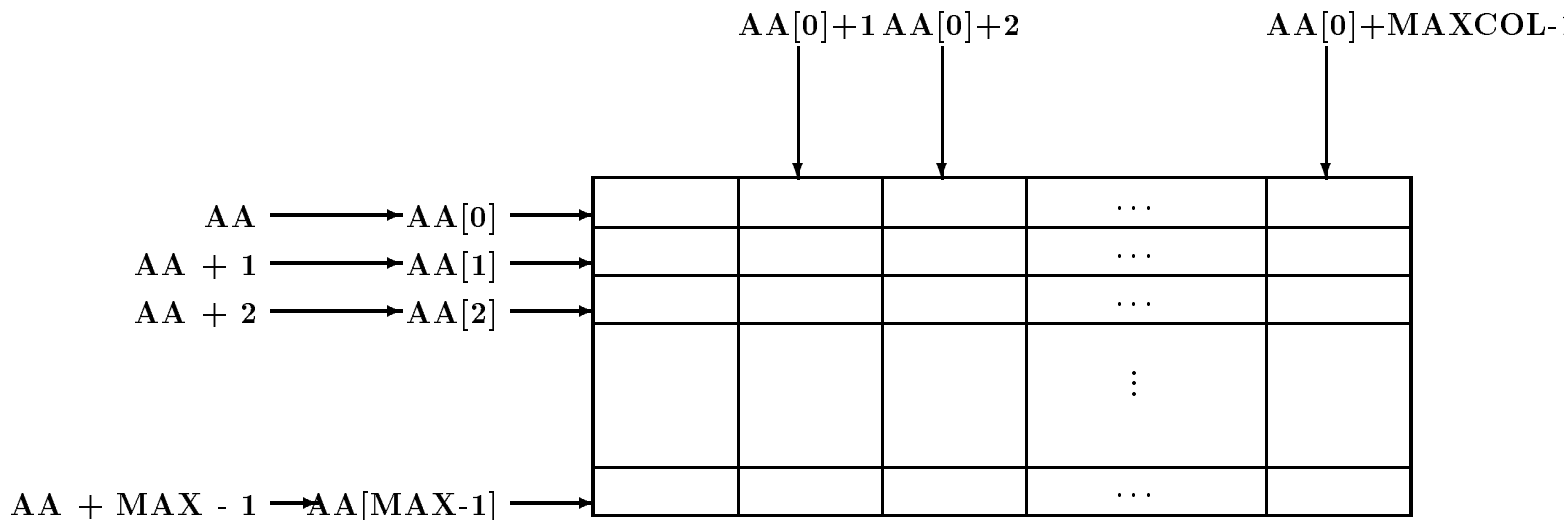
Figure 9.11: Pointers and Two Dimensional Arrays

making it point to the next element. The value of `intptr` is printed as it is incremented. Observe that it prints the address of each element of the array in one row, and proceeds to the next row in sequence. This shows that arrays are stored in row major form.

Finally, the function, `print2aray()` is used to print the two dimensional array in rows and columns. The output of a sample run is shown below.

```
***2D Arrays, Pointers ***

array (row) pointer a = 65474, *a = 65474
a + 0 = 65474

*a + COL * 0 + 0 = 65474; intptr = 65474
&a[0][0] = 65474
a[0][0] = 12; *(*(a + 0) + 0) = 12

*a + COL * 0 + 1 = 65476; intptr = 65476
&a[0][1] = 65476
a[0][1] = 24; *(*(a + 0) + 1) = 24

*a + COL * 0 + 2 = 65478; intptr = 65478
&a[0][2] = 65478
a[0][2] = 29; *(*(a + 0) + 2) = 29

a + 1 = 65480
*a + COL * 1 + 0 = 65480; intptr = 65480
&a[1][0] = 65480
a[1][0] = 23; *(*(a + 1) + 0) = 23
```

```
/*    File: ar2ptr.c
      Other Source Files: ar2util.c
      Header Files: ar2def.h, ar2util.h
      Program shows relations between arrays and pointers for 2 dimensional
      arrays.
*/

#include <stdio.h>
#define ROW 2
#define COL 3
print2aray(int x[][COL], int r, int c);

main()
{    int i, j, *intptr, a[ROW][COL] =
                        { {12, 24, 29}, {23, 57, 19} };

     printf("***2D Arrays, Pointers ***\n\n");
     intptr = *a;
     printf("array (row) pointer a = %u,   *a = %u\n", a, *a);
     for (i = 0; i < ROW; i++) {
          printf("a + %d = %u\n", i, a + i);
          for (j = 0; j < COL; j++) {
               printf("*a + COL * %d + %d = %u; intptr = %u\n",
                         i, j, *a + COL * i + j, intptr);
               printf("&a[%d][%d] = %u\n", i, j, &a[i][j]);

               printf("a[%d][%d] = %d; *(*(a + %d) + %d) = %d\n",
                         i, j, a[i][j],
                         i, j, *(*(a + i) + j));
               intptr++;
          }
     }
     print2aray(a, ROW, COL);
}
     /*    This Function prints a two dimensional integer array. */
     print2aray(int x[][COL], int r, int c)
     { int i, j;

          printf("\nThe two dimensional array is:\n\n");
          for (i = 0; i < r; i++) {
               for (j = 0; j < c; j++)
                    printf("%d  ", x[i][j]);
               printf("\n");
          }
     }
```

Figure 9.12: Program Illustrating 2D Array Pointers

```
*a + COL * 1 + 1 = 65482; intptr = 65482
&a[1][1] = 65482
a[1][1] = 57; *(*(a + 1) + 1) = 57


*a + COL * 1 + 2 = 65484; intptr = 65484
&a[1][2] = 65484
a[1][2] = 19; *(*(a + 1) + 2) = 19


The two dimensional array is:
 12   24   29
 23   57   19
```

As we mentioned in the last section, when a two dimensional array is passed to a function, the parameter declaration in the function **must** include the number of columns. We can now see why this is so. The number of columns in a row specifies the size of each row in the array of rows. Since the passed parameter is a pointer to a row object, it can be incremented and dereferenced, as shown in Table 9.3, to access the elements of the two dimensional array. The compiler must know the size of the row in order to be able to increment the pointer to the next row.

As we stated earlier, multi-dimensional arrays are arrays of arrays just like two dimensional arrays. An n dimensional array is an array of $n-1$ dimensional arrays. The same general approach applies as for two dimensional arrays. When passing an n dimensional array, the declaration of the formal parameter must specify all index ranges except for the first index.

As was seen in the program in Figure 9.12, multi-dimensional arrays may also be initialized in declarations by specifying constant initializers within braces. Each initializer must be appropriate for the corresponding lower dimensional array. For example, a two dimensional array may be initialized as follows:

```
int x[2][3] = { {10, 23}, {0, 12, 17} };
```

The array has two elements, each of which is an array of three elements. The first initializer initializes the first row of **x**. Since only the first two elements of the row are specified, the third element is zero. The second element initializes the second row. Thus, **x** is initialized to the array:

```
10    23    0
0     12    17
```

9.3   Arrays of Strings

Besides data base applications, another common application of two dimensional arrays is to store an array of strings. In this section we see how an array of strings can be declared and operations such as reading, printing and sorting can be performed on them.

A string is an array of characters; so, an array of strings is an array of arrays of characters. Of course, the maximum size is the same for all the strings stored in a two dimensional array. We can declare a two dimensional character array of **MAX** strings of size **SIZE** as follows:

```
char names[MAX][SIZE];
```
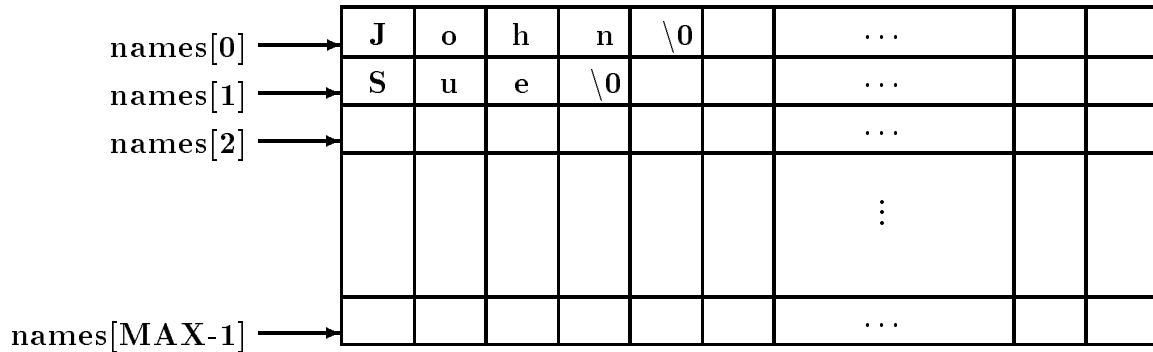
Figure 9.13: An Array of Strings

Since **names** is an array of character arrays, **names[i]** is the ith character array, i.e. it points to the ith character array or string, and may be used as a string of maximum size **SIZE - 1**. As usual with strings, a **NULL** character must terminate each character string in the array. We can think of an array of strings as a table of strings, where each row of the table is a string as seen in Figure 9.13.

We will need an array of strings in our next task to read strings, store them in an array, and print them.

NAMES: Read and store a set of strings. Print the strings.

We can store a string into **names[i]** by reading a string using **gets()** or by copying one into it using **strcpy()**. Since our task is to read strings, we will use **gets()**. The algorithm is simple:

```
while array not exhausted and not end of file,
     read a string into an array element
print out the strings in the array of strings
```

We will organize the program in several source files since we will be using some of the functions in several example programs. The program driver and the header file are shown in Figure 9.14.

The program reads character strings into an array, in this case, names. The program can, of course, serve to read in any strings. The **for** loop in **main()** reads strings into an array using **gets()** to read a string into **names[n]**, the nth row of the array. That is, the string is stored where **names[n]** points to. The variable **n** keeps track of the number of names read. The loop is terminated either if the number of names equals **MAX**, or when **gets()** returns **NULL** indicating end of file has been reached. Next, the program calls on **printstrtab()** to print the names stored in the two dimensional array, **names**. The arguments passed are the array of strings and the number of strings, **n**.

The function, **printstrtab()** is included in the file **strtab.c** and its prototype is included in the file **strtab.h**. Remember, the second range of the two dimensional array of strings must be specified in the formal parameter definition, otherwise the number of columns in a row are unknown and the function cannot access successive rows correctly. A sample interaction for the compiled and linked program is shown below:

Sample Session:

```
***Table of Strings - Names***
```

```
/*   File: strtab.h */
#define SIZE 31      /* maximum size of a name plus a NULL */
void printstrtab(char strtab[][SIZE], int n);

    /*   File: names.c
         Other Source Files: strtab.c
         Header Files: strtab.h
         This program reads a set of names or strings into a two
         dimensional array. It then prints out the names.
    */

    #include <stdio.h>
    #define MAX 10
    #include "strtab.h"

    main()
    {   int n;                       /* number of names */
        char names[MAX][SIZE];    /* 2-d array of names */

        printf("***Table of Strings - Names***\n\n");
        printf("Enter one name per line, EOF to terminate\n");

        for (n = 0; (n < MAX) && gets(names[n]); n++)
             ;

        if (n == MAX)
             printf("\n**Table full - input terminated\n");
        printstrtab(names, n);
    }

    /*   File: strtab.c */
    #include <stdio.h>
    #include "strtab.h"

    /*   Prints n strings in the array strtab[][]. */
    void printstrtab(char strtab[][SIZE], int n)
    {   int k;

        printf("Names are:\n");
        for (k = 0; k < n; k++)
             puts(strtab[k]);
    }
```

Figure 9.14: Code for String Table Driver

```
Enter one name per line, EOF to terminate
```
John Smith
David Jones
Helen Peterson
Maria Schell
^D
```
Names are:
John Smith
David Jones
Helen Peterson
Maria Schell
```

9.3.1   String Sorting and Searching

Our next couple of tasks are simple and build on the last one. In one task we search (sequentially) for a string and in another we sort a set of strings.

SRCHSTR: Search for a key string in a set of strings.

We will use a function, `srchstr()`, to search for a string in an array of a specified size. The function returns either a valid index where the string is found or it returns -1 to indicate failure. The algorithm is simple enough and the implementation of a test program driver is shown in Figure 9.15.

The file `strtab.h` includes the prototypes for functions in file `strtab.c`. Observe the initialization of the two dimensional array `names[][]` using constant initializers written in braces separated by commas. Each initializer initializes a one dimensional string array written as a string constant. The program calls `srchstrtab()` searching for the string `"John Smith"`, and prints the returned index value.

As we have seen in Chapter 8, the library function `strcmp()` is used to compare two strings. The function returns zero if the argument strings are equal. A sequential search process for strings is easily developed by modifying the sequential search function of Chapter 10 replacing the equality operator with the function `strcmp()` to compare two strings.

In the function, `srchstrtab()`, we compare each string in the array with the desired string until we either find a match or the array is exhausted. The function call requires the name of the array of strings, the number of valid elements in the array, and the item to be searched for. For example, suppose we wish to search for a string, `key`, in the array, `names` with `n` string elements, then, the function call would be:

```
k = srchstrtab(names, n, key);
```

The value returned is assigned to an integer variable, `k`. If successful, `srchstrtab()` returns the index where the string was found; otherwise, it returns -1. The function is shown in Figure 9.16. In the for loop, the string that `strtab[i]` points to is compared with the string that `key` points to. If they are equal, `strcmp()` returns zero and the value of `i` is returned by the function. Otherwise, `i` is incremented, and the process is repeated. The loop continues until the valid array is exhausted, in which case -1 is returned. Again, the formal parameter definition for the two dimensional array, `x`, requires the size of the second dimension, `SIZE`. A sample run of the program is shown below:

```
/*   File: strsrch.c
     Other Source Files: strtab.c
     Header Files: strtab.h
     This program searches for a string (key) in a set of strings
     in a two dimensional array. It prints the index where key is found.
     It prints -1, if the string is not found.
*/

#include <stdio.h>
#define MAX 10
#include "strtab.h"

main()
{    int k;
     char names[MAX][SIZE] = { "John Jones", "Sheila Smith",
                                   "John Smith", "Helen Kent"};

     printf("***String Search for John Smith***\n\n");
     k = srchstrtab(names, 4, "John Smith");
     printf("John Smith found at index %d\n", k);
}
```

Figure 9.15: Driver for String Search Program

```
/*   File: strtab.c - continued */
#include <string.h>
/*   Searches a string table strtab[][] of size n for a string key. */
int srchstrtab(char strtab[][SIZE], int n, char key[])
{    int i;

     for (i = 0; i < n; i++)
          if (strcmp(strtab[i], key) == 0)
               return i;
     return -1;
}
```

Figure 9.16: Code for srchstrtab()

```
/*   File: strsort.c
     Other Source Files: strtab.c
     Header Files: strtab.h
     This program sorts a set of strings in a two dimensional array.
     It prints the unsorted and the sorted set of strings.
*/

#include <stdio.h>
#define MAX 10
#include "strtab.h"

main()
{    int n;
     char names[MAX][SIZE] = { "John Jones", "Sheila Smith",
                                   "John Smith", "Helen Kent"};

     printf("***String Array - unsorted and sorted***\n\n");
     printf("Unsorted ");
     printstrtab(names, 4);

     sortstrtab(names, 4);
     printf("Sorted ");
     printstrtab(names, 4);
}
```

Figure 9.17: Driver for Sorting Strings Program

```
***String Search for John Smith***

John Smith found at index 2
```

Our next task calls for sorting a set of strings.

SORTSTR: Sort a set of strings. Print strings in unsorted and in sorted order.

The algorithm is again very simple and we implement it in the program driver. The driver simply calls on sortstrtab() to sort the strings and prints the strings, first unsorted and then sorted. A prototype for sortstrtab() is included in file strtab.h and the driver is shown in Figure 9.17. An array of strings is initialized in the declaration and the unsorted array is printed. Then, the array is sorted, and the sorted array is printed.

Sorting of an array of strings is equally straight forward. Let us assume, the array of strings is to be sorted in increasing ASCII order, i.e. a is less than b, b is less than c, A is less than a, and so on. We will use the selection sort algorithm from Chapter 10. Two nested loops are needed; the inner loop moves the largest string in an array of some effective size to the highest index in the array, and the outer loop repeats the process with a decremented effective size until the effective size is one. The function is included in file strtab.c and shown in Figure 9.18. The function

```
/*     File: strtab.c - continued */
/*     Sorts an array of strings. The number of strings in the
       array is lim.
*/
void sortstrtab(char strtab[][SIZE], int lim)
{    int i, eff_size, maxpos = 0;
     char tmp[SIZE];

     for (eff_size = lim; eff_size > 1; eff_size--) {
          for (i = 0; i < eff_size; i++)

               if (strcmp(strtab[i], strtab[maxpos]) > 0)
                    maxpos = i;

          strcpy(tmp, strtab[maxpos]);
          strcpy(strtab[maxpos], strtab[eff_size-1]);
          strcpy(strtab[eff_size - 1], tmp);
     }
}
```

Figure 9.18: Code for `sortstrtab()`

is similar to the numeric selection sort function, except that we now use `strcmp()` to compare strings and `strcpy()` to swap strings. A sample session is shown below:

```
***String Array - unsorted and sorted***

Unsorted Names are:
John Jones
Sheila Smith
John Smith
Helen Kent

Sorted Names are:
Helen Kent
John Jones
John Smith
Sheila Smith
```

In our example program, the entire strings are compared. If we wish to sort by last name, we could modify our function to find and compare only the last names.

9.4   Arrays of Pointers

As seen in the last example, sorting an array of strings requires swapping the strings which can require copying a lot of data. For efficiency, it is better to avoid actual swapping of data whenever a data item is large, such as a string or an entire data base record. In addition, arrays may be needed in more than one order; for example, we may need an exam scores array sorted by Id number and by weighted scores; or, we may need strings in both an unsorted form and a sorted form. In either of these cases, we must either keep two copies of the data, each sorted differently, or find a more efficient way to store the data structure. The solution is to use pointers to elements of the array and swap pointers. Consider some examples:

```
int data1, data2, *ptr1, *ptr2, *save;

data1 = 100; data2 = 200;
ptr1 = &data1; ptr2 = &data2;
```

We could swap the values of the data and store the swapped values in `data1` and `data2` or we could simply swap the values of the pointers:

```
save = ptr1;
ptr1 = ptr2;
ptr2 = save;
```

We have not changed the values in `data1` and `data2`; but `ptr1` now accesses `data2` and `ptr2` access `data1`. We have swapped the pointer values so they point to objects in a different order. We can apply the same idea to strings:

```
char name1[] = "John";
char name2[] = "Dave";
char *p1, *p2, *save;

p1 = name1;
p2 = name2;
```

Pointers `p1` and `p2` point to strings `name1` and `name2`. We can now swap the pointer values so `p1` and `p2` point to `name2` and `name1`, respectively.

In general, an array of pointers can be used to point to an array of data items with each element of the pointer array pointing to an element of the data array. Data items can be accessed either directly in the data array, or indirectly by dereferencing the elements of the pointer array. The advantage of a pointer array is that the pointers can be reordered in any manner without moving the data items. For example, the pointer array can be reordered so that the successive elements of the pointer array point to data items in sorted order without moving the data items. Reordering pointers is relatively fast compared to reordering large data items such as data records or strings. This approach saves a lot of time, with the additional advantage that the data items remain available in the original order. Let us see how we might implement such a scheme.

STRPTRS: Given an array of strings, use pointers to order the strings in sorted form, leaving the array unchanged.

We will use an array of character pointers to point to the strings declared as follows:
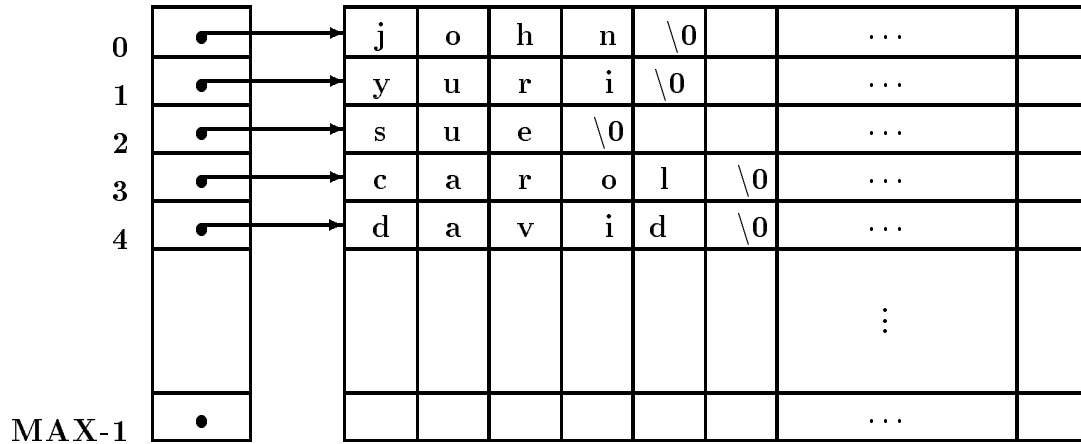
captionUnsorted Pointers to Strings

```
char * nameptr[MAX];
```

The array, `nameptr[]`, is an array of size `MAX`, and each element of the array is a character pointer. It is then possible to assign character pointer values to the elements of the array; for example:

```
nameptr[i] = "John Smith";
```

The string `"John Smith"` is placed somewhere in memory by the compiler and the pointer to the string constant is then assigned to `nameptr[i]`. It is also possible to assign the value of any string pointer to `nameptr[i]`; for example, if `s` is a string, then it is possible to assign the pointer value `s` to `nameptr[i]`:

```
nameptr[i] = s;
```

In particular, we can read strings into a two dimensional array, `names[][]`, and assign each string pointer, `names[i]` to the ith element of the pointer array, `nameptr[]`:

```
for (i = 0; i < MAX && gets(names[i]); i++)
    nameptr[i] = names[i];
```

The strings can then be accessed either by `names[i]` or by `nameptr[i]` as seen in Figure 9.4. We can then reorder the pointers in `nameptr[]` so that they successively point to the strings in sorted order as seen in Figure 9.4. We can then print the strings in the original order by accessing them through `names[i]` and print the strings in sorted order by accessing them through `nameptr[i]`. Here is the algorithm:

```
while not end of file and array not exhausted,
    read a string
    store it in an array of strings and
    assign the string to an element of a pointer array

access the array of strings and print them out
access the array of pointers and print strings that point to
```

	j	o	h	n	\0		...	
	y	u	r	i	\0		...	
	s	u	e	\0			...	
	c	a	r	o	l	\0	...	
	d	a	v	i	d	\0	...	

0, 1, 2, 3, 4 ... MAX-1
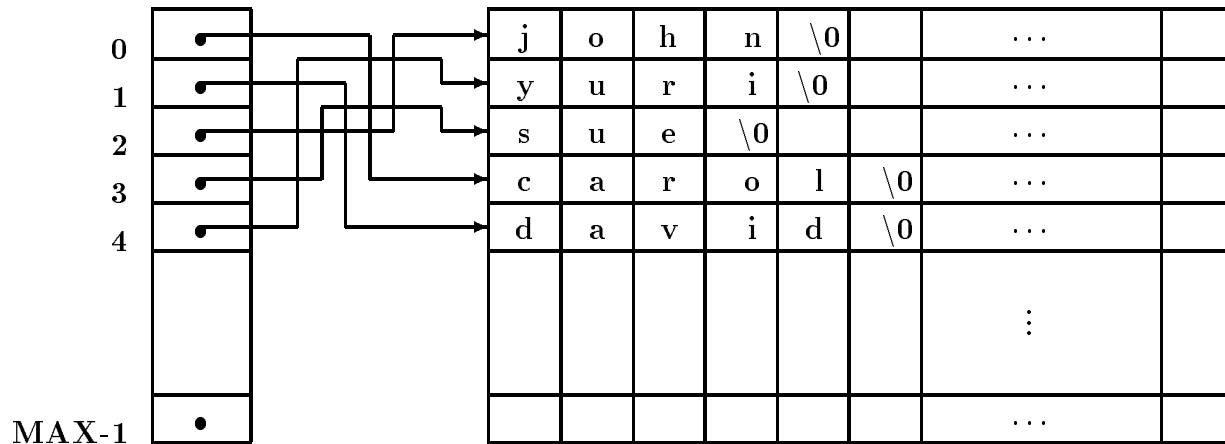
captionSorted Pointers to Strings

The program driver, including a prototype for `sortptrs()` is shown in Figure 9.19. It declares a two dimensional array of strings, `names[][]`, and an array of character pointers, `nameptr[]`. It then reads strings into `names[][]`, and assigns each string pointer `names[i]` to `nameptr[i]`. The function `sortptrs()` is then called to reorder `nameptr[]` so the successive pointers of the array point to the strings in sorted order. Finally, strings are printed in original unsorted order by accessing them through `names[i]` and in sorted order by accessing them through `nameptr[i]`.

The function `sortptrs()` uses the selection sort algorithm modified to access data items through pointers. It repeatedly moves the pointer to the largest string to the highest index of an effective array. The implementation of sorting using pointers to strings is shown in Figure 9.20. The algorithm determines `maxpos`, the index of the pointer to the largest string. The pointer at `maxpos` is then moved to the highest index in the effective array. The array size is then reduced, etc.

Sample Session:

```
***Arrays of Pointers - Sorting by Pointers***

Enter one name per line, EOF to terminate
john
yuri
sue
carol
david
^D

The unsorted names are:
john
yuri
sue
carol
david
```

```
/*    File: ptraray.c
      This program uses an array of pointers. Elements of the array
      point to strings. The pointers are reordered so that they
      point to the strings in sorted order. Unsorted and sorted
      strings are printed out.
*/

#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define MAX  10           /* max number of names */
#define SIZE 31           /* size of names plus one for NULL */
void sortptrs(char * nameptr[], int n);

main()
{     int i;              /* counter */
      int n;              /* number of names read */
      char names[MAX][SIZE]; /* 2-d array of names */
      char *nameptr[MAX]; /* array of ptrs - used to point to names */

      printf("***Arrays of Pointers - Sorting by Pointers***\n\n");
      /* read the names into the 2-d array */
      printf("Enter one name per line, ");
      printf("EOF to terminate\n");

      for (n = 0; gets(names[n]) && n < MAX; n++)
          nameptr[n] = names[n];   /* assign string pointer */
                                   /* to a char pointer in the */
                                   /* array of pointers. */

      if (n == MAX)
          printf("\n***Only %d names allowed***\n", MAX);

      printf("The unsorted names are:\n");
      /* print the names */
      for (i = 0; i < n; i++)
          puts(names[i]);            /* access names in stored array.*/

      sortptrs(nameptr, n);          /* sort pointers */
      printf("The sorted names are:\n");
      for (i = 0; i < n; i++)        /* print sorted names, */
          puts(nameptr[i]);          /* accessed via array of pointers. */
}
```

Figure 9.19: Driver for Sorting Pointer Array Program

```
/*   File: ptraray.c  - continued */
/*   The elements of the array of pointers nameptr[] point to
     strings. The pointer array is reordered so the pointers
     point to strings in sorted order. The function uses selection
     sort algorithm.
*/

void sortptrs(char * nameptr[], int n)
{    int i, eff_size, maxpos = 0;
     char *tmpptr;

     for (eff_size = n; eff_size > 1; eff_size--) {
          for (i = 0; i < eff_size; i++)

               if (strcmp(nameptr[i],nameptr[maxpos]) > 0)
                    maxpos = i;

          tmpptr = nameptr[maxpos];
          nameptr[maxpos] = nameptr[eff_size-1];
          nameptr[eff_size-1] = tmpptr;
     }
}
```

Figure 9.20: Code for sortptrs()

```
The sorted names are:
carol
david
john
sue
yuri
```

Reordering of pointers, so they point to data items in sorted order, is referred to as *sorting by pointers*. When the data items are large, such as data records or strings, this is the preferred way of sorting because it is far more efficient to move pointers than it is to move entire data records.

9.5   An Example: Linear Algebraic Equations

As our final example program using two dimensional arrays in this chapter, we develop a program to solve systems of simultaneous linear equations. A set of linear algebraic equations, also called simultaneous equations, occur in a variety of mathematical applications in science, engineering, economics, and social sciences. Examples include: electronic circuit analysis, econometric analysis, structural analysis, etc. In the most general case, the number of equations, n, may be different from the number of unknowns, m; thus, it may not be possible to find a unique solution. However, if n equals m, there is a good chance of finding a unique solution for the unknowns.

Our next task is to solve a set of linear algebraic equations, assuming that the number of equations equals the number of unknowns:

LINEQNS: Read the coefficients and the right hand side values for a set of linear equations; solve the equations for the unknowns.

The solution of a set of linear equations is fairly complex. We will first review the process of solution and then develop the algorithm in small parts. As we develop parts of the algorithm, we will implement these parts as functions. The driver will just read the coefficients, call on a function to solve the equations, and call a function to print the solution.

Let us start with an example of a set of three simultaneous equations in three unknowns: x_1, x_2, and x_3.

$$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 = 6$$

$$2 \cdot x_1 + 3 \cdot x_2 + 1 \cdot x_3 = 6$$

$$1 \cdot x_1 + 0 \cdot x_2 + 2 \cdot x_3 = 3$$

We can use arrays to represent such a set of equations; a two dimensional array to store the coefficients, a one dimensional array to store the values of the unknowns when solved, and another one dimensional array to store the values on the right hand side. Later, we will include the right hand side values as an additional column in the two dimensional array of coefficients. Each row of the two dimensional array stores the coefficients of one of the equations. Since the array index in C starts at 0, we will assume the unknowns are the elements `x[0]`, `x[1]`, and `x[2]`. Similarly, the elements in row zero are the coefficients in the equation number 0, the elements in row one are for equation number one, and so forth.

Then using arrays, a general set of n linear algebraic equations with m unknowns may be expressed as shown below:

```
a[0][0] * x[0] +  a[0][1]*x[1] + ...+  a[0][m - 1] * x[m - 1]  =  y[0]
a[1][0] * x[0] +  a[1][1]*x[1] + ...+  a[1][m - 1] * x[m - 1]  =  y[1]
...
a[n-1][0]*x[0] +     ...               + a[n-1][m - 1]*x[m - 1]  =  y[n-1]
```

The unknowns and the right hand side are assumed to be elements of one dimensional arrays:
`x[0]`, `x[1]`,..., `x[m - 1]` and `y[0]`, `y[1]`,..., `y[n - 1]`, respectively. The coefficients are
assumed to be elements of a two dimensional array: `a[i][j]` for $i = 0, \ldots, n - 1$ and $j = 0, \ldots, m - 1$. The coefficients of each equation correspond to a row of the array. For our discussion
in this section, we assume that m equals n. With this assumption, it is possible to find a unique
solution of these equations unless the equations are linearly dependent, i.e. some equations are
linear combinations of others.

A common method for solving such equations is called the *Gaussian elimination* method.
The method eliminates (i.e. makes zero) all coefficients below the main diagonal of the two
dimensional array. It does so by adding multiples of some equations to others in a systematic way.
The elimination makes the array of new coefficients have an upper triangular form since the lower
triangular coefficients are all zero.

The modified equivalent set of n equations in $m = n$ unknowns in the upper triangular form
have the appearance shown below:

```
a[0][0]*x[0]+  a[0][1]*x[1] +    ...          +   a[0][n-1]  *x[n-1] =   y[0]
               a[1][1]*x[1] +    ...          +   a[1][n-1]  *x[n-1] =   y[1]
                                 a[2][2]*x[2]..+  a[2][n-1]  *x[n-1] =   y[2]
                                                  a[n-1][n-1]*x[n-1]=   y[n-1]
```

The upper triangular equations can be solved by *back substitution*. Back substitution first solves
the last equation which has only one unknown, `x[n-1]`. It is easily solved for this value — `x[n-1]`
`= y[n-1]/a[n-1][n-1]`. The next to the last equation may then be solved — since `x[n-1]` has
been determined already, this value is substituted in the equation, and this equation has again only
one unknown, `x[n-2]`. The unknown, `x[n-2]`, is solved for, and the process continues backward
to the next higher equation. At each stage, the values of the unknowns solved for in the previous
equations are substituted in the new equation leaving only one unknown. In this manner, each
equation has only one unknown which is easily solved for.

Let us take a simple example to see how the process works. For the equations:

```
1 * x[0] + 2 * x[1] + 3 * x[2] = 6
2 * x[0] + 3 * x[1] + 1 * x[2] = 6
1 * x[0] + 0 * x[1] + 2 * x[2] = 3
```

We first reduce to zero the coefficients in the first column below the main diagonal (i.e. array index
zero). If the first equation is multiplied by -2 and added to the second equation, the coefficient in
the second row and first column will be zero:

```
1 * x[0] + 2 * x[1] + 3 * x[2] =  6
0 * x[0] - 1 * x[1] - 5 * x[2] = -6
1 * x[0] + 0 * x[1] + 2 * x[2] =  3
```

Similarly, if the first equation is multiplied by -1 and added to the third equation, the coefficient
in the third row and first column will be zero:

```
1 * x[0] + 2 * x[1] + 3 * x[2] =  6
0 * x[0] - 1 * x[1] - 5 * x[2] = -6
0 * x[0] - 2 * x[1] - 1 * x[2] = -3
```

Coefficients in the first column below the main diagonal are now all zero, so we do the same for the second column. In this case, the second equation is multiplied by a multiplier and added to equations below the second; thus, multiplying the second equation by -2 and adding to the third makes the coefficient in the second column zero:

```
1 * x[0] + 2 * x[1] + 3 * x[2] = 6
0 * x[0] - 1 * x[1] - 5 * x[2] = -6
0 * x[0] + 0 * x[1] + 9 * x[2] = 9
```

We now have equivalent equations with an upper triangular form for the non-zero coefficients. The equations can be solved backwards — the last equation gives us `x[2]` = 1. Substituting the value of `x[2]` in the next to the last equation and solving for `x[1]` gives us `x[1]` = 1. Finally, substituting `x[2]` and `x[1]` in the first equation gives us `x[0]` = 1.

From the above discussion, we can see that a general algorithm involves two steps: modify the coefficients of the equations to an upper triangular form, and solve the equations by back substitution.

Let us first consider the process of modifying the equations to an upper triangular form. Since only the coefficients and the right hand side values are involved in the computations that modify the equations to upper triangular form, we can work with these items stored in an array with n rows and $n + 1$ columns (the extra column contains the right hand side values).

Let us assume the process has already reduced to zero the first $k - 1$ columns below the main diagonal, storing the modified new values of the elements in the same elements of the array. Now, it is time to reduce the kth lower column to zero (by lower column, we mean the part of the column below the main diagonal). The situation is shown in below:

```
a[0][0]    a[0][1]    ...           a[0][k]        ...         a[0][n]
0          a[1][1]    ...           a[1][k]        ...         a[1][n]
0          0          a[2][2]...    a[2][k]        ...         a[2][n]
...        ...        ...           ...            ...         ...
0          0          0...          a[k][k]        a[k][k+1].. a[k][n]
0          0          0...          a[k+1][k]      ...         a[k+1][n]
0          0          0...          a[k+2][k]      ...         a[k+2][n]
...        ...        ...           ...            ...         ...
0          0          0...          a[n-1][k]      ...         a[n-1][n]
```

The nth column represents the right hand side values with `a[i][n]` equal to `y[i]`. We multiply the kth row by an appropriate multiplier and add it to each row with index greater than k. Assuming that `a[k][k]` is non-zero, the kth row multiplier for addition to the ith row $(i > k)$ is:

```
-a[i][k] / a[k][k]
```

The kth row multiplied by the above multiplier and added to the ith row will make the new `a[i][k]` zero. The following loop will reduce to zero the lower k^{th} column:

```
/*    Algorithm: process_column
      Reduces lower column k to zero.
*/
for (i = k + 1; i < n; i++) {        /* process rows k+1 to n-1 */
  m = - a[i][k] / a[k][k]            /* multiplier for ith row */

  for (j = k; j <= n; j++)           /* 0 thru k-1 cols. are zero */
    a[i][j] += m * a[k][j];          /* add kth row times m */
                                     /* to ith row. */

}
```

However, before we can use the above loop to reduce the lower k^{th} column to zero, we must make sure that a[k][k] is non-zero. If the current a[k][k] is zero, all we need to do is exchange this k^{th} row with any higher indexed row with a non-zero element in the kth column. After the exchange of the two rows, the new a[k][k] will be non-zero. The above loop is then used to reduce the lower k^{th} column to zero. The non-zero element, a[k][k] used in the multiplier is called a *pivot*.

So, there are two steps involved in modifying the equations to upper triangular form: for each row find a pivot, and reduce the corresponding lower column to zero. If a non-zero pivot element is not found, then one or more equations are linear combinations of others, the equations are called *linearly dependent*, and they cannot be solved.

Figures 9.22 and 9.23 show the set of functions that convert the first n rows and columns of an array to an upper triangular form. These and other functions use a user defined type, status, with possible values ERROR returned if there is an error, and OK returned otherwise. The type status is defined as follows:

```
typedef enum {ERROR, OK} status;
```

We also assume a maximum of MAX equations, so the two dimensional array must have MAX rows and MAX+1 columns. Figure 9.21 includes the header file with the defines and function prototypes used in the program. Since precision is important in these computations, we have used formal parameters of type double. The two dimensional arrays can store coefficients for a maximum of MAX equations (rows) and have MAX + 1 columns to accommodate the right hand side values.

The function uptriangle() transforms coefficients of the equations to an upper triangular form. For each k from 0 through n-1, it calls findpivot() to find the pivot in the kth column. If no pivot is found, findpivot() will return an ERROR (findpivot() is called even for the $(n-1)^{st}$ column even though there is no lower $(n-1)^{st}$ column to test if a[n-1][n-1] is zero). If findpivot() returns OK, then uptriangle() calls process_col() to reduce the lower k^{th} column to zero. We have included debug statements in process_col() to help track the process. The function pr2adbl() prints the two dimensional array — we will soon write this function.

The function findpivot() calls on function findnonzero() to find a non-zero pivot in column k if a[k][k] is zero. If a pivot is found, it swaps the appropriate rows and returns OK. Otherwise, it reurns ERROR. The function findnonzero() merely scans the lower column k for a non-zero element. It either returns the row in which it finds a non-zero element or it returns -1 if no such element is found. Rows of the array are swapped by the function swaprows() which also includes a debug statement to prints the row indices of the rows being swapped.

When uptriangle() returns with OK status, the array will be in upper triangular form. The next step in solving the equations is to employ back substitution to find the values of the unknowns.

```
/*    File: gauss.h */
typedef enum {ERROR, OK} status;
#define DEBUG
#define MAX 10                /* maximum number of equations */


status uptriangle(double a[][MAX + 1], int n);
void process_col(double a[][MAX + 1], int k, int n);
status findpivot(double a[][MAX + 1], int k, int n);
int findnonzero(double a[][MAX + 1], int k, int n);
void swaprows(double a[][MAX + 1], int k, int j, int n);
status gauss(double a[][MAX + 1], double x[], int n);
int getcoeffs(double a[][MAX + 1]);
void pr2adbl(double a[][MAX + 1], int n);
void pr1adbl(double x[], int n);
```

Figure 9.21: Header File for Gauss Functions

We now examine the back substitution process. As we saw earlier, we must solve equations backwards starting at index $n-1$ and proceeding to index 0. The ith equation in upper triangular form looks like this:

```
a[i][i]*x[i] + a[i][i+1]*x[i+1] + ... + a[i][n-1]*x[n-1] = a[i][n]
```

Recall, in our representation, the right hand side is the nth column of the two dimensional array. For each index i, we must sum all contributions from those unknowns already solved for, i.e. those x[i] with index greater than i. This is the following sum:

```
sum = a[i][i+1]*x[i+1] + ... + a[i][n-1]*x[n-1]
```

We then subtract this sum from the right hand side, a[i][n], and divide the result by a[i][i] to determine the solution for x[i]. The algorithm is shown below:

```
/*    Algorithm: Back_Substitution */
for (i = n - 1; i >= 0;  i--) {          /* go backwards */
      sum = 0;

      for (j = i + 1; j <= n - 1; j++)   /* sum all contributions from */
            sum += a[i][j] * x[j];       /* x[j] with j > i */
      x[i] = (a[i][n] - sum) / a[i][i];  /* solve for x[i] */
}
```

We can now write the function gauss() that solves a set of equations by the Gaussian elimination method which first calls on uptriangle() to convert the coefficients to upper triangular form. If this succeeds, then back substitution is carried out to find the solutions. As with other functions, gauss() returns OK if successful, and ERROR otherwise. The code is shown in Figure 9.24. The code is straight forward. It incorporates the back substitution algorithm after the function call to uptriangle(). If the function call returns ERROR, the equations cannot be solved

```c
/* File: gauss.c */
#include <stdio.h>
#include "gauss.h"

/*   Implements the Gauss method to transform equations to
     an upper triangular form.
*/
status uptriangle(double a[][MAX + 1], int n)
{    int i, j, k;

     for (k = 0; k < n; k++) {
          if (findpivot(a, k, n) == OK)
               process_col(a, k, n);
          else
               return ERROR;
     }
     return OK;
}

/*   Zeros out coefficients in column k below the main diagonal. */
void process_col(double a[][MAX + 1], int k, int n)
{    int i, j;
     double m;

     for (i = k + 1; i < n; i++) {
          m = -a[i][k] / a[k][k];
          for (j = k; j <= n; j++)
               a[i][j] += m * a[k][j];
          #ifdef DEBUG
               printf("Multiplier for row %d is %6.2f\n", i, m);
               pr2adbl(a, n);
          #endif
     }
}
```

Figure 9.22: Code for Functions to do Gaussian Elimination

```
/*    Finds a non-zero pivot element in column k and row with
      index >= k.
*/
status findpivot(double a[][MAX + 1], int k, int n)
{     int j;
      void swaprows();

      if (a[k][k] == 0) {
            j = findnonzero(a, k, n);
            if (j < 0)
                  return ERROR;
            else
                  swaprows(a, k, j, n);
            #ifdef DEBUG
                  printf("Rows %d and %d swapped\n", k, j);
            #endif
      }
      return OK;
}


/*    Scans the rows with index >= k for the first non-zero element
      in the kth column of the array 'a' of size n.
*/
int findnonzero(double a[][MAX + 1], int k, int n)
{     int i;

      for (i = k; i < n; i++)
            if (a[i][k])
                  return(i);
      return(-1);
}


/*    Swaps the kth and the jth rows in the array 'a' with n rows. */
void swaprows(double a[][MAX + 1], int k, int j, int n)
{     int i;
      double temp;

      for (i = k; i <= n; i++) {
            temp = a[k][i];
            a[k][i] = a[j][i];
            a[j][i] = temp;
      }
}
```

Figure 9.23: Code for Functions to do Gaussian Elimination — continued

```
/*    File: gauss.c - continued */
/*    Transforms equations to upper triangular form using Gauss
      method. Then, solves equations, one at a time.
*/
status gauss(double a[][MAX + 1], double x[], int n)
{    int i, j;
     double sum;

     if (uptriangle(a, n) == ERROR) {
         printf("Dependent equations - cannot be solved\n");
         return ERROR;
     }

     for (i = n - 1; i >= 0;  i--) {
         sum = 0;

         for (j = i + 1; j <= n - 1; j++)
             sum += a[i][j] * x[j];

         if (a[i][i])
             x[i] = (a[i][n] - sum) / a[i][i];
         else
             return ERROR;

     }
     return OK;
}
```

Figure 9.24: Code for gauss()

and `gauss()` returns `ERROR`. Otherwise, `gauss()` proceeds with back substitution and stores the result in the array `x[]`. Since all `a[i][i]` must be non-zero at this point, we do not really need to test if `a[i][i]` is zero before using it as a divisor; however, we do so as an added precaution.

We are almost ready to use the function `gauss()` in a program. Before we can do so; however, we need some utility functions to read and print data. Here are the descriptions of these functions:

`getcoeffs()`: reads the coefficients and the right hand side values into an array; it returns the number of equations.

`pr2adbl()`: prints an array with n rows and $n+1$ columns.

`pr1adbl()`: prints a solution array.

All these functions use data of type double. The code is shown in Figure 9.25.

Finally, we are ready to write a program driver as shown in Figure 9.26. The driver first reads coefficients and the right hand side values for a set of equations and then calls on `gauss()` to solve the equations. During the debug phase, both the original data and the transformed upper triangular version are printed. Finally, if the equations are solved with success, the solution is printed. Otherwise, an error message is printed. During debugging, the macro `DEBUG` is defined in `gauss.h` so that we can track the process. The program loops as long as there are equations to be solved. In each case, it gets coefficients using `getcoeffs()` and solves them using `gauss()`. During debug, the program uses `pr2adbl()` to print the original array and the array after gauss transformation. If the solution is possible, the program prints the solution array using `pr1adbl()`. Here are several example equation solutions:

Sample Session:

```
***Simultaneous Equations - Gauss Elimination Method***

Number of equations, zero to quit:  2
Type coefficients and right side of each row
Row 0:  1 3 2
Row 1:  3 5 2

Original equations are:
   1.00   3.00   2.00
   3.00   5.00   2.00

Multiplier for row 1 is -3.00
   1.00    3.00    2.00
   0.00   -4.00   -4.00

Equations after Gauss Transformation are:
   1.00    3.00    2.00
   0.00   -4.00   -4.00

Solution is:
-1.00
1.00
```

```
/*    File: gauss.c - continued */
/*    Function gets the coefficients and the right hand side of equations.
*/
int getcoeffs(double a[][MAX + 1])
{    int i, j, n;

     printf("Number of equations, zero to quit: ");
     scanf("%d", &n);

     if (n)
          printf("Type coefficients and right side of each row\n");

     for (i = 0; i < n; i++) {
          printf("Row %d: ", i);

          for (j = 0; j <= n; j++)
               scanf("%lf", &a[i][j]);
     }
     return n;
}


/*    Prints coefficients and right side of equations */
void pr2adbl(double a[][MAX + 1], int n)
{    int i, j;

     for (i = 0; i < n; i++) {

          for (j = 0; j <= n; j++)
               printf("%10.2f  ", a[i][j]);

          printf("\n");
     }
}


/*    Prints the solution array */
void pr1adbl(double x[], int n)
{    int i;

     for (i = 0; i < n; i++)
          printf("%10.2f\n", x[i]);
}
```

Figure 9.25: Code for Utility Functions for Gauss Program

```
/*   File: gauss.c
     Header Files: gauss.h
     This program solves a number of simultaneous linear algebraic
     equations using the Gauss elimination method. The process repeats
     itself until number of equations is zero.
*/

main()
{    double a[MAX][MAX + 1];   /* coefficients and right hand side */
     double x[MAX];            /* solution */
     int n;                    /* number of equations */
     status soln;              /* status of solution, OK or ERROR */

     printf("***Simultaneous Equations***\n\n");
     while (n = getcoeffs(a)) {
          printf("\nOriginal equations are:\n");
          #ifdef DEBUG
               pr2adbl(a, n);
          #endif

          soln = gauss(a, x, n);
          #ifdef DEBUG
               printf("\nEquations after Gauss Transformation are:\n");
               pr2adbl(a, n);
          #endif

          if (soln == OK) {
               printf("\nSolution is:\n");
               pr1adbl(x, n);
          }

          else printf("Equations cannot be solved\n");
     }
}
```

Figure 9.26: Driver Program for Gaussian Elimination

```
Number of equations, zero to quit:   3
Type coefficients and right side of each row
Row 0:   1 2 3 4
Row 1:   4 3 2 1
Row 2:   0 7 2 5

Original equations are:
    1.00   2.00   3.00   4.00
    4.00   3.00   2.00   1.00
    0.00   7.00   2.00   5.00

Multiplier for row 1 is -4.00
    1.00    2.00    3.00     4.00
    0.00   -5.00  -10.00   -15.00
    0.00    7.00    2.00     5.00

Multiplier for row 2 is -0.00
     1.00    2.00    3.00     4.00
     0.00   -5.00  -10.00   -15.00
    -0.00    7.00    2.00     5.00

Multiplier for row 2 is 1.40
    1.00    2.00    3.00     4.00
    0.00   -5.00  -10.00   -15.00
    0.00    0.00  -12.00   -16.00

Equations after Gauss Transformation are:
    1.00    2.00    3.00     4.00
    0.00   -5.00  -10.00   -15.00
    0.00    0.00  -12.00   -16.00

Solution is:
-0.67
0.33
1.33


Number of equations, zero to quit:   3
Type coefficients and right side of each row
Row 0:   1 2 3 4
Row 1:   2 4 6 8
Row 2:   3 1 7 9

Original equations are:
    1.00   2.00   3.00   4.00
    2.00   4.00   6.00   8.00
    3.00   1.00   7.00   9.00

Multiplier for row 1 is -2.00
```

```
   1.00   2.00   3.00   4.00
   0.00   0.00   0.00   0.00
   3.00   1.00   7.00   9.00
Multiplier for row 2 is -3.00
   1.00    2.00    3.00    4.00
   0.00    0.00    0.00    0.00
   0.00   -5.00   -2.00   -3.00
Rows 1 and 2 swapped

Multiplier for row 2 is 0.00
   1.00    2.00    3.00    4.00
   0.00   -5.00   -2.00   -3.00
   0.00   -0.00   -0.00   -0.00
Dependent equations - cannot be solved

Equations after Gauss Transformation are:
   1.00    2.00    3.00    4.00
   0.00   -5.00   -2.00   -3.00
   0.00   -0.00   -0.00   -0.00
Equations cannot be solved
Number of equations, zero to quit:   0
```

The first two sets of equations are solvable; the last set is not because the second equation in the last set is a multiple of the first equation. Thus these equations are linearly dependent and they cannot be solved uniquely. In this case, after the $zero^{th}$ lower column is reduced to zero, a[1][1] is zero. A pivot is found in row 2, rows 1 and 2 are swapped, and lower column 1 is reduced to zero. However, a[2][2] is now zero, and there is no unique way to solve these equations.

If the coefficients are such that the equations are almost but not quite linearly dependent, the solution can be quite imprecise. An improvement in precision may be obtained by using an element with the *largest* absolute value as the pivot. Implementation of an improved version of the method is left as an exercise.

9.6   Common Errors

1. Failure to specify ranges of smaller dimensional arrays in declaration of formal parameters. All but the range of the first dimension must be given in a formal parameter declaration. Example:

   ```
   init2(int aray2[][])
   {
        ...
   }
   ```

   Error! aray2 is a pointer to a two dimensional array, i.e. it points to an object that is a one-dimensional array, aray2[0]. Without a knowledge of the size of the object, aray2[0],

it is not possible to access `aray2[1]`, `aray2[2]`, etc. Consequently, one must specify the number of integer objects in `aray2[0]`:

```
init2(int aray2[][COLS])
{ ...
}
```

Correct! `aray2[0]` has `COLS` objects. It is possible to advance the pointer, `aray2` correctly to the next row, etc.

2. Failure to pass arguments correctly in function calls:

```
init2(aray2[MAX][COLS]);
init2(aray2[][COLS]);
init2(aray2[][]);
```

All of the above are errors. A two dimensional array *name* is passed in a function call:

```
init2(aray2);
```

3. Confusion between pointers to different types of objects. For example, in the above, `aray2` points to an *array* object, `aray2[0]`, whereas `aray2[0]` points to an `int` object. The expression `aray2 + 1` points to `aray2[1]`, whereas `aray2[0] + 1` points to `aray2[0][1]`. In the first case the pointer is increased by `COLS` integer objects, whereas in the second case the pointer is increased by one integer object.

4. Confusion between arrays of character strings and arrays of character pointers:

```
char table[MAX][SIZE], *ptraray[MAX];
```

The first declares table to be a two dimensional array that can be used to store an array of strings, one each in `table[0]`, `table[1]`, `table[i]`, etc. The second declares `ptraray` to be an array, each element of which is a `char *`. Read the declaration from the end: `[MAX]` says it is an array with `MAX` elements; `ptraray` is the name of the array; `char *` says each element of the array is a `char *`. Properly initialized with strings stored in `table[][]`, `table[i]` can point to a string. Properly initialized with pointers to strings, `ptraray[i]` can also point to a string. However, `table[MAX][SIZE]` provides memory space for the strings, whereas `ptraray[MAX]` provides memory space only for pointers to strings. Both pointers may be used in a like manner:

```
puts(table[i]);
puts(ptraray[i]);
```

They will both print the strings pointed to by the pointers.

9.7 Summary

In this chapter we have seen that, in C, the concept of an array can be extended to arrays of multi-dimensions. In particular, a two dimensional array is represented as a one dimensional array, each of whose elements, themselves, are one dimensional arrays, i.e. and array or arrays. Similarly, a three dimensional array is an array whose elements are each 2 dimensional arrays (an array of arrays of arrays). We have seen how such arrays are declared within programs and how they are organized in memory (row major order). We have seen how we can access the elements of multi dimensional arrays using the subscripting notation and the correspondence between this notation and pointer values. Because for higher dimensional arrays, the pointer expressions may get complicated and confusing, in general, most programs use the subscripting notations for arrays of two dimensions or more. We have also shown that when passing arrays to functions, the size of all dimensions beyond the first must be specified in the formal parameter list of the function so that the location of all elements can be calculated by the compiler.

Throughout the chapter we have seen applications for which a two dimensional data structure provides a convenient and compact way of organizing information. These have included data base applications, such as our payroll and student test score examples, as well as using two dimensional arrays to store an array of strings. We have seen how we can then use this later data structure to search and sort arrays of strings, and have shown that for this data type, as well as other large data types, it is often more efficient to work with arrays of pointers when reordering such data structures.

Finally, we have developed a rather large application using 2D arrays — solutions to simultaneous linear equations usinf Gaussian elimination. This is one algorithm for doing computations in the realm of linear algebra; several additional examples common in engineering problems are presented in Chapter 15.

One last point to remember about multi-dimensional arrays: this data structure is a very useful way to organize a large collection of data into one common data structure; however, all of the data items in this structure must be of the same type. In the next chapter we will see another compound data type provided in C which does not have such a restriction — the *structure*.

9.8   Exercises

Given the following declaration:

```
int x[10][20];
```

 Explain what each of the following represent:

 1. `x`

 2. `x + i`

 3. `*(x + i)`

 4. `*(x + i) + j`

 5. `*(*(x + i) + j)`

 6. `x[0]`

 7. `x[i]`

 8. `x[i] + j`

 9. `*(x[i] + j)`

 Find and correct errors if any. What does the program do in each case?

10.
```
main()
{     int x[5][10];

      init(x[][]);
}

void init(int a[][])
{     int i, j;

      for (i = 0; i < 10; i++)
          for (j = 0; j < 5; j++)
              a[i][j] = 0;
}
```

11.
```
main()
{     int x[5][10];

      init(x[][]);
}

void init(int *a)
{     int i, j;
```

```
        for (i = 0; i < 10; i++)
            for (j = 0; j < 5; j++) {
                *a = 0;
                a++;
            }
    }

12. main()
    {   char s[5][100];

        read_strings(s);
        print_strings(s);
    }

    read_strings(char s[][100])
    {
        for (i = 0; i < 5; i++) {
            gets(*s);
            s++;
        }
    }

    print_strings(char s[][100])
    {
        while (*s) {
            puts(s);
            s++;
        }
    }
```

9.9   Problems

1. Read id numbers, project scores, and exam scores in a two dimensional array from a file. Compute the averages of each project and exam scores; compute and store the weighted average of the scores for each id number.

2. Repeat 1, but sort and print the two dimensional array by weighted average in decreasing order. Sort and print the array by id numbers in increasing order. Use an array of pointers to sort.

3. Repeat 2, but plot the frequency of each weighted score.

4. Combine 1-3 into a menu-driven program with the following options: read names, id numbers, and scores from a file; add scores for a new project or exam; save scores in a file; change existing scores for a project or an exam for specified id numbers; delete a data record; add a data record; compute averages; sort scores in ascending or descending order by a primary key, e.g. id numbers, weighted scores, etc.; compute weighted average; plot frequency of weighted scores; help; quit.

5. Write a function that uses binary search algorithm to search an array of strings.

6. Write a function that sorts strings by selection sort in either increasing or decreasing order.

7. Write a program that takes a string and breaks it up into individual words and stores them.

8. Repeat 7 and keep track of word lengths. Display the frequency of different word lengths.

9. Repeat 7, but store only new words that occur in a string. If the word has already been stored, ignore it.

10. Write a function that checks if the set of words in a string, s, represents a subset of the set of words in a second string, t. That is, the words of s are all contained in t, with t possibly containing additional words.

11. Write a menu-driven spell check program with the following options: read a dictionary from a file; spell check a text file; add to dictionary; delete from dictionary; display text buffer; save text buffer; help; quit.

    The dictionary should be kept sorted at all times and searched using binary search. Use an array of pointers to sort when new entries are inserted. In the spell check option, the program reads in lines of text from a file. Each word in a line is checked with the dictionary. If the word is present in the dictionary, it is ignored. Otherwise, the user is asked to make a decision: replace the word or add it to the dictionary. Either replace the word with a new word in the line or add the word to dictionary. Each corrected line is appended to a text buffer. At the quit command, the user is alerted if the text buffer has not been saved.

12. Write a simple macro processor. It reads lines from a source file. Ignoring leading white space, each line is examined to see if it is a control line starting with a symbol # and followed by a word "define". If it is, store the defined identifier and the replacement string. Each line is examined for the possible occurrence of each and every defined identifier; if a defined

identifier occurs in a line, replace it with the replacement string. The modified line must be examined again to see if a defined identifier exists; if so, replace the identifier with a string, etc.

13. Write a lexical scanner, `scan()`, which calls `nexttok()` of Problem 42 to get the next token from a string. Each new token or symbol of type identifier, integer, or float that is found is stored in a symbol table. The token is inserted in the table if and only if it was not already present. A second array keeps track of the type of each token stored at an index. The function `scan()` uses `srcharay()` to search the array of tokens, uses `inserttok()` to insert a token in an array, and uses `inserttype()` to insert type of a token.

    The function `scan()` returns a token in a string, type of the token, and index where the token is stored in the array. If the array is filled, a message saying so must be displayed.

    Write a program driver to read strings repeatedly. For each string, call `scan()` to get a token. As `scan()` returns, print the token, its type, and index. Repeat until an end of string token is reached. When the end of file is encountered, print each of the tokens, its type, and index.

14. Write routines for drawing lines and rectangles. Write a program that draws a specified composite figure using a character `'*'`. Allow the user to specify additions to the figure and display the figure when the user requests.

15. Modify 14 to a menu-driven program that allows: draw horizontal and vertical lines, horizontally oriented rectangles, filled rectangles, display figure, help, and quit.

16. Write a program that plays a game of tic-tac-toe with the user. The game has three rows and three columns. A player wins when he succeeds in filling a row or a column or a diagonal with his mark, `'0'`. The program uses `'*'`. Write and use the following functions:

    `init_board()`: initialize the board

    `display_board()`: dispays the board

    `enter_move()`: for user to enter a move in row and col

    `state_of_game()`: test state, finish or continue

17. Modify the Gauss Method so that a pivot with the largest magnitude is used in converting the array of coefficients to an upper triangular form.

18. Modify 17 to a menu-driven program that allows the following commands: Get coefficients, display coefficients, solve equations, display solution, verify solution, help, and quit. Write and use functions `get_coeffs()`, `display_coeffs()`, `solve_eqns()`, `display_soln()`, `verify_soln()`, `help()`.

19. Modify 18 so that the input data is in the form:

    ```
    a00 x0 + a01 x1 + a02 x2 = b1
    ```

20. Modify 19 so that display coefficients displays equations in the above form.

21. Write a simple menu driven editor which allows the following commands: text insert, display text, delete text, delete lines, insert lines, find string, find word, replace string, replace word, help, and quit. A window should display part of the text when requested.

PART II

Chapter 10

Sorting and Searching

One very common application for computers is storing and retrieving information. For example, the telephone company stores information such as the names, addresses and phone numbers of its customers. When you dial directory assistance to get the phone number for someone, the operator must look up that particular piece of information from among all of data that has been stored. Taken together, all of this information is one form of a *data base* which is organized as a collection of *records*. Each record consists of several *fields*, each containing one piece of information, such as the name, address, phone number, id number, social security number or part number, etc..

As the amount of information to be stored and accessed becomes very large, the computer proves to be a useful tool to assist in this task. Over the years, as computers have been applied to these types of tasks, many techniques and algorithms have been developed to efficiently maintain and process information in data bases. In this chapter, we will develop and implement some of the simpler instances of these algorithms. The processes of "looking up" a particular data record in the data base is called *searching*. We will look at two different search algorithms; one very easy to implement, but inefficient, the other much more efficient. As we will see, in order to do an efficient search in a data base, the records must be maintained in some *order*. For example, consider the task of finding the phone number in the phone book of someone whose name you know, as opposed to trying to find the name of someone whose phone number you know in the same book.

The process of ordering the records in a data base is called *sorting*. We will discuss three sorting algorithms and their implementation in this chapter, as well. Sorting and searching together constitute a major area of study in computational methods. We present some of these methods here to introduce this area of computing as well as to make use of some of the programming techniques we have developed in previous chapters.

As we develop these algorithms, we use a very simple data base of records consisting of single integers only. We conclude the chapter by applying the searching and sorting techniques to our payroll data base with records consisting of multiple numeric fields. In Chapter 9 we will see how these same algorithms can be applied to string data types described in Chapter 11.

10.1   Finding a Data Item — The Search Problem

Suppose we have a collection of data items of some specific type (e.g. integers), and we wish to determine if a particular data item is in the collection. The particular data item we want to find
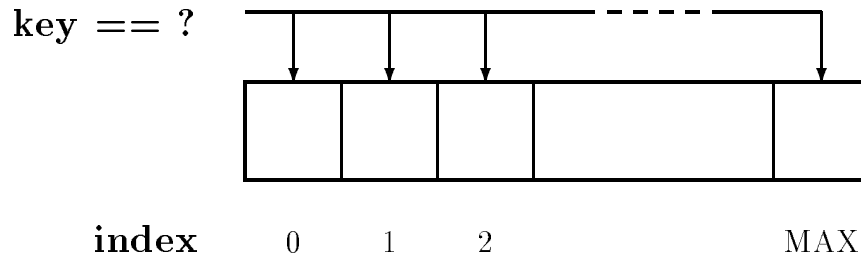
Figure 10.1: Sequential Search

is called the *key* and out task is to search the records in the data base to find one which "matches" the key.

The first decision we must make is how to represent the collection of data items. In Chapter 7 we saw a data structure which could hold a collection of data items all of the same type: the array. So we can consider our data base of integer values to be stored in an array. Our task then becomes:

Task:

SRCH0: Search an array for an index where the key is located; if key is not present, print a message. Repeat until an end of file is entered for the key.

In this task, we choose to return the index where the key is located because this index will allow us to retrieve the entire record in the case where our array is part of a database. The simplest approach to determine if a key is present in an array is to make an exhaustive search of the array. Start with the first element, if the key matches, we are done; otherwise move on to the next element and compare the key, and so on. We simply traverse the array in sequence from the first element to the last as shown in Figure 10.1. Each element is compared to the key. If the key is found in the array, the corresponding array index is returned. If the item is not found in the array, an invalid index, say -1, is returned. This type of search is called *Sequential Search* or *Linear Search* because we sequentially examine the elements of the array. In the worst case, the number of elements that must be compared with the key is linearly proportional to the size of the array.

Linear search is not the most efficient way to search for an item in a collection of items; however, it is very simple to implement. Moreover, if the array elements are arranged in random order, it is the only reasonable way to search. In addition, efficiency becomes important only in large arrays; if the array is small, there aren't many elements to search and the amount of time it takes is not even noticed by the user. Thus, for many situations, linear search is a perfectly valid approach. Here is a linear search algorithm which returns the index in the array where key is found or -1 if key is not found in the array:

```
initialize index i to 0
traverse the array until exhausted
    if array[i] matches key
        return i;
return -1.
```

```
/*   File: sortsrch.c  */
#include <stdio.h>
#include "sortsrch.h"
#define DEBUG
/*
     Linear or sequential search of an array x[] of size lim for
     an item key.
*/
int seqsrch(int x[], int lim, int key)
{    int i;

     for (i = 0; i < lim; i++)
          if (x[i] == key)
               return(i);
     return(-1);
}
```

Figure 10.2: Code for the function `seqsrch()`

```
/*   File: sortsrch.h
     This file contains prototypes for sort and search functions.
*/
int seqsrch(int x[], int lim, int key);
```

Figure 10.3: Initial contents of `sortsrch.h`

We will implement the algorithm for search as a function, `seqsrch()` since searching an array may be required in many programs, and a function incorporating linear search can be used for many applications. The function is passed the array of integers, `x[]`, the number of elements in the array, `lim`, and the key to search for, `key`. The function is shown in Figure 10.2.

The loop compares each element of the array with `key`. If an element with the same value is found at an index `i`, `i` is returned. The loop terminates when the array limit is reached; in which case, no element equal to `key` was found in the array, and -1 is returned. If there is more than one element in the array with the same value as `key`, the function terminates the search as soon as the first element is found, returning its index.

We have defined `DEBUG` in `sortsrch.c` so that debug statements we may add to the code will be compiled. We have also included file `sortsrch.h` in `sortsrch.c` which, as shown in Figure 10.3 contains the prototypes for functions defined in `sortsrch.c` since some of the functions defined in `sortsrch.c` may be used by other functions to be written in the file. We will continue to add more functions to `sortsrch.c` and corresponding prototypes to `sortsrch.h` as we proceed through this chapter; however, we will not always show the additions of prototypes to `sortsrch.h`.

Our task now requires us to write a simple driver to repeatedly call the function, `seqsrch()`. For simplicity, we will declare an initialized array. The driver uses a function, `pr_aray_line()`, to

print an array with ten elements per line to save space. Figure 10.4 shows the program driver.

The driver first prints an initialized array, `id[]`, and then searches for items entered by the user. If an item is found in the array, its index is printed. Otherwise, a message is printed. The function, `pr_array_line()` is included in `sortsrch.c` and its prototype in `sortsrch.h`. These additions are shown in Figure 10.5.

The function prints at successive elements on one line until the array index modulo 10 is zero when it prints a newline and continues. A sample session for the program `srcharay.c` is shown below:

```
***Sequential Search***

The id array is:
45 67 12 34 25 39

Type an integer, EOF to quit:   23
Item 23 is not in the array

Type an integer, EOF to quit:   12
Item 12 is found at index 2

Type an integer, EOF to quit:   45
Item 45 is found at index 0

Type an integer, EOF to quit:   ^D
```

10.2   Improving Search — Sorting the Data

As we mentioned above, linear search may be useful when the number of elements is small; however, when there are many data items in the array, linear search may require a long time to find the element matching the key. In the case where such an element is not in the data base, we must search the entire collection to find that out. Consider the phone book again. When we want to look up someone's phone number, we do not start at the beginning of the book and look line by line until we find the name. Instead, we make use of the fact that the data items are sorted by name in the phone book. (Of course, if we had a phone number and wanted to find the name, we would have to resort to linear search — we do not often do that).

Before we develop an algorithm to conduct a more efficient search of sorted data, we first describe several algorithms which will sort an array of data. There are numerous ways to sort data, some more suitable than others for different applications. In this section we will describe three different standard algorithms: selection sort, bubble sort, and insertion sort.

10.2.1   Selection Sort

The idea of selection sort is rather simple: we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index

```
/*    File:srcharay.c
      Other Source Files: sortsrch.c
      Header Files: sortsrch.h
      This program searches an array sequentially for items typed in
      by the user. It prints out the array index where an item is found,
      or else prints a message.
*/

#include <stdio.h>
#include "sortsrch.h"

main()
{     int id[] = {45, 67, 12, 34, 25, 39};
      int n, i;

      printf("***Sequential Search***\n\n");
      printf("The array is:\n");
      pr_aray_line(id, 6);
      printf("Type an integer, EOF to quit: ");

      while (scanf("%d", &n) != EOF) {
          i = seqsrch(id, 6, n);

          if (i >= 0)
              printf("Item %d is found at index %d\n", n, i);
          else
              printf("Item %d is not in the array\n", n);

          printf("Type an integer, EOF to quit: ");
      }
}
```

Figure 10.4: Driver to test `seqsrch()`

```
/* File: sortsrch.c - continued */
/* Prints an array horizontally. */
void pr_aray_line(int x[], int lim)
{    int i;

     for (i = 0; i < lim; i++) {
          if (i % 10 == 0)
               printf("\n");
          printf("%d  ", x[i]);
     }

     printf("\n");
}


/* File: sortsrch.h - continued */
void pr_aray_line(int x[], int lim);
```

Figure 10.5: Adding the code for `pr_aray_line()`

position. We can do this by swapping the element at the highest index and the largest element. We then reduce the *effective size* of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

For example, consider the following array, shown with array elements in sequence separated by commas:

    63,   75,   **90**,   12,   **27**

The leftmost element is at index zero, and the rightmost element is at the highest array index, in our case, 4 (the effective size of our array is 5). The largest element in this effective array (index 0-4) is at index 2. We have shown the largest element and the one at the highest index in **bold**. We then swap the element at index 2 with that at index 4. The result is:

    63,   **75**,   27,   **12**,   90

We reduce the effective size of the array to 4, making the highest index in the effective array now 3. The largest element in this effective array (index 0-3) is at index 1, so we swap elements at index 1 and 3 (in **bold**):

    63,   12,   **27**,   75,   90

The next two steps give us:

    27,   **12**,   63,   75,   90
    12,   27,   63,   75,   90

The last effective array has only one element and needs no sorting. The entire array is now sorted. The algorithm for an array, x, with `lim` elements is easy to write down:

```
for (eff_size = lim; eff_size > 1; eff_size--)
    find maxpos, the location of the largest element in the effective
        array: index 0 to eff_size - 1
    swap elements of x at index maxpos and index eff_size - 1
```

The implementation of the selection sort algorithm in C, together with a driver program is shown in Figure 10.6.

Sample Session:

```
Original array:
63 75 90 12 27
Sorted array:
12 27 63 75 90
```

The driver prints the array, calls `selection_sort()` to sort the array, and prints the sorted array. The code for `selection_sort()` follows the algorithm exactly; it calls `get_maxpos()` to get the index of the largest element in an array of a specified size. Once `maxpos` is found, the element at that index is swapped with the element at index `eff_size-1`, using the temporary variable, `tmp`.

We may be concerned about the efficiency of our algorithm and its implementation as a program. The efficiency of an algorithm depends on the number of major computations involved in performing the algorithm. The efficiency of the program depends on that of the algorithm and the efficiency of the code implementing the algorithm. Notice we included the code for swapping array elements within the loop in `selection_sort` rather than calling a function to perform this operation. A function call requires added processing time in order to store argument values, transfer program control, and retrieve the returned value. When a function call is in a loop that may be executed many times, the extra processing time may become significant. Thus, if the array to be sorted is quite large, we can improve program efficiency by eliminating a function call to swap data elements. Similarly, we may include the code for `get_maxpos()` in `selection_sort()`:

```c
void selection_sort(int x[], int lim)
{    int i, eff_size, maxpos, tmp;

     for (eff_size = lim; eff_size > 1; eff_size--) {
          for (i = 0; i < eff_size; i++)
               maxpos = x[i] > x[maxpos] ? i : maxpos;
          tmp = x[maxpos];
          x[maxpos] = x[eff_size - 1];
          x[eff_size - 1] = tmp;
     }
}
```

10.2.2   Bubble Sort

An alternate way of putting the largest element at the highest index in the array uses an algorithm called *bubble sort*. While this method is neither as efficient, nor as straightforward, as selection sort, it is popularly used to illustrate sorting. We include it here as an alternate method.

Like selection sort, the idea of bubble sort is to repeatedly move the largest element to the highest index position of the array. As in selection sort, each iteration reduces the effective size of the array. The two algorithms differ in how this is done. Rather than search the entire effective

```
/* File: select.c
   This program implements selection sort.
*/
#include <stdio.h>
#define MAX 10
void selection_sort(int x[], int lim);
int get_maxpos(int x[], int lim);
void print_aray(int x[], int lim);

main()
{   int scores[MAX] = {63, 75, 90, 12, 27};

    printf("Original array:\n");
    print_aray(scores, 5);
    selection_sort(scores, 5);
    printf("Sorted array:\n");
    print_aray(scores, 5);
}
/* Selection sort function for an array x[] with lim elements. */
void selection_sort(int x[], int lim)
{   int eff_size, maxpos, tmp;

    for (eff_size = lim; eff_size > 1; eff_size--) {
      maxpos = get_maxpos(x, eff_size);
      tmp = x[maxpos];
      x[maxpos] = x[eff_size - 1];
      x[eff_size - 1] = tmp;
    }
}
/* Function returns the index of the largest element in the array x[]. */
int get_maxpos(int x[], int eff_size)
{   int i, maxpos = 0;

    for (i = 0; i < eff_size; i++)
      maxpos = x[i] > x[maxpos] ? i : maxpos;
    return maxpos;
}
/* Function prints an integer array of size lim. */
void print_aray(int x[], int lim)
{   int i;

    for (i = 0; i < lim; i++)
      printf("%d  ", x[i]);
}
```

Figure 10.6: Sorting an array using Selection Sort

array to find the largest element, bubble sort focuses on successive adjacent pairs of elements in the array, compares them, and either swaps them or not. In either case, after such a step, the larger of the two elements will be in the higher index position. The focus then moves to the next higher position, and the process is repeated. When the focus reaches the end of the effective array, the largest element will have "bubbled" from whatever its original position to the highest index position in the effective array.

For example, consider the array:

45, **67**, 12, 34, 25, 39

In the first step, the focus is on the first two elements (in **bold**) which are compared and swapped, if necessary. In this case, since the element at index 1 is larger than the one at index 0, no swap takes place.

45, **67**, **12**, 34, 25, 39

Then the focus move to the elements at index 1 and 2 which are compared and swapped, if necessary. In our example, 67 is larger than 12 so the two elements are swapped. The result is that the largest of the first three elements is now at index 2.

45, 12, **67**, **34**, 25, 39

The process is repeated until the focus moves to the end of the array, at which point the largest of all the elements ends up at the highest possible index. The remaining steps and result are:

45, 12, 34, **67**, **25**, 39
45, 12, 34, 25, **67**, **39**
45, 12, 34, 25, 39, 67

The largest element has bubbled to the top index of the array. In general, a bubble step is performed by the loop:

```
for (k = 0; k < eff_size - 1; k++)
    if (x[k] > x[k + 1])
        swaparay(x, k, k + 1);
```

The loop compares all adjacent elements at index k and k + 1. If they are not in the correct order, they are swapped. One complete bubble step moves the largest element to the last position, which is the correct position for that element in the final sorted array. The effective size of the array is reduced by one and the process repeated until the effective size becomes one. Each bubble step moves the largest element in the effective array to the highest index of the effective array.

The code implementing this algorithm is the function, `bubblesort()` shown in Figure 10.7. The function repeats bubble steps, using the function `bubblemax()`, as many times as the size of the array. This function is passed the array name and the size of the effective array. The size of the effective array is the original size reduced by one after each step. Thus, if the initial size of the array to be sorted is `lim`, the size of each successive effective array is `lim`, `lim -1`, `lim - 2`, etc. We have included a debug statement in `bubblesort()` to trace the bubble process after each bubble step. The function, `bubblemax()`, compares adjacent elements of an array of the specified size in sequence and swaps them if necessary. The function is shown in Figure 10.8 together with the function, `swaparay()` to swap elements in an array. All these functions are included in file, `sortsrch.c`, and their prototypes are included in file,`sortsrch.h`, also shown in the Figure. It should be clear that bubble sort is not as efficient as selection sort. There is a great deal of swapping required in bubble sort to "bubble" the largest element to the highest index; where in selection sort, it is done by a single swap. On the other hand, if the data is mostly sorted, then bubble sort can be made more efficient.

```
/* File: sortsrch.c - continued */
/* Sorts an array x of size lim using bubble sort. */
void bubblesort(int x[], int lim)
{    int i;

     for (i = 0; i < lim; i++)
     {    bubblemax(x, lim - i);   /* effective array size is lim - i */
          #ifdef DEBUG               /* debug statement */
               printf("Effective array of size %d: \n", lim - i);
               pr_aray_line(x, lim - i);
          #endif
     }
}
```

Figure 10.7: Code for bubble sort

```
/* File: sortsrch.c - continued */
/* bubbles the next largest element through the array x */
void bubblemax(int x[], int eff_size)
{    int k;

     for (k = 0; k < eff_size - 1; k++)
          if (x[k] > x[k + 1])
               swaparay(x, k, k + 1);
}

/* File: sortsrch.c - continued */
/* swaps elements i and j of array x */
void swaparay(int x[], int i, int j)
{    int temp;

     temp = x[i];
     x[i] = x[j];
     x[j] = temp;
}


/* File: sortsrch.h - continued */
void bubblesort(int x[], int lim);
void bubblemax(int x[], int eff_size);
void swaparay(int x[], int i, int j);
```

Figure 10.8: Code for `bubblemax()`

```
/*   File: bsrtaray.c
     Other Source Files: sortsrch.c
     Header Files: sortsrch.h
     This program uses bubble sort to sort an array of
     integers. It prints the unsorted and the sorted
     arrays. It also prints a trace at each bubble step to
     show the bubble process.
*/

#include <stdio.h>
#define DEBUG
#include "sortsrch.h"

main()
{    int id[] = {45, 67, 12, 34, 25, 39};

     printf("***Bubble Sort***\n\n");
     printf("Unsorted array: \n");
     pr_aray_line(id, 6);

     bubblesort(id, 6);
     printf("Sorted array: \n");
     pr_aray_line(id, 6);
}
```

Figure 10.9: Driver to test bubble sort

To illustrate the operation of bubble sort, we now write a program driver to exercise bubble sort shown in Figure 10.9. It uses `bubblesort()` on the same array used in our search example above. The initialized unsorted array is printed; then the array is sorted and printed. Each bubble step is explicitly shown by a debug statement. Note that DEBUG is defined during program development and removed when the program is debugged.

Sample Session:

```
***Bubble Sort***

Unsorted array:
45 67 12 34 25 39

Effective array of size 6:
45 12 34 25 39 67

Effective array of size 5:
12 34 25 39 45
```

```
/*   File: sortsrch.c - continued */
/*   Bubble sort in a single function */
void bsrtfnc(int x[], int lim)
{    int i, k, temp;

     for (i = 0; i < lim; i++) {
          for (k = 0; k < lim - i - 1; k++)

               if (x[k] > x[k + 1]) {
                    temp = x[k];
                    x[k] = x[k+1];
                    x[k+1] = temp;
               }

               #ifdef DEBUG
                    printf("Effective array of size %d: \n", lim - i);
                    pr_aray_line(x, lim - i);
               #endif
     }
}

/* File: sortsrch.h - continued */
void bsrtfnc(int x[], int lim);
```

Figure 10.10: Code for one function bubble sort

```
Effective array of size 4:
12 25 34 39

Effective array of size 3:
12 25 34

Effective array of size 2:
12 25

Effective array of size 1:
12

Sorted array:
12 25 34 39 45 67
```

There are several ways to improve the bubble sort algorithm. First, a single function should incorporate the entire algorithm (Figure 10.10). The time overhead of a function call in a loop can be quite large if the array is large.

Next, a minor point: since an array of one element is already sorted, at most $n - 1$ bubbling

```
/*    File: sortsrch.c - continued */
/*    Bubble sort function which terminates if an array is sorted. */
#include "tfdef.h"        /* defines TRUE and FALSE */
void bsort(int x[], int lim)
{    int i, k, temp, swap = TRUE;

     for (i = 0; swap && i < lim - 1; i++)
     {    swap = FALSE;

          for (k = 0; k < lim - i - 1; k++)
               if (x[k] > x[k + 1])
               {    temp = x[k];
                    x[k] = x[k+1];
                    x[k+1] = temp;
                    swap = TRUE;
               }

          #ifdef DEBUG
               printf("Effective array of size %d: \n", lim - i);
               pr_aray_line(x, lim - i);
          #endif
     }
}

/* File: sortsrch.h - continued */
void bsort(int x[], int lim);
```

Figure 10.11: An improved bubble sort

steps are needed for an array of size n. The first for loop need be executed no more than `lim - 1` times. More important, if the entire array is sorted at some time in the process, no further processing is needed. An array is sorted if no elements are swapped in a bubble step. We will use a flag to keep track of any swapping. Figure 10.11 shows the revised code.

We include a file, `tfdef.h`, that defines `TRUE` and `FALSE`. In the function, we use a flag, `swap`, to keep track of any swapping in the bubble step. For each bubble step, we initially assume `swap` is `FALSE`. If there is any swapping in the bubble step, we set the flag to `TRUE`. The sort process repeats as long as swap is `TRUE`. To get the process started, swap is initialized to `TRUE`.

These improvements may be important for large arrays. If an array is sorted after the first few steps, the process can be terminated with a saving in computation time. The program, `bsrtaray.c`, can be modified to use the above `bsort()` function instead of `bubblesort()` function. A sample output of such a modified program is shown below.

Sample Session (Modified bsrtaray.c):

```
***Bubble Sort***
```

```
Unsorted array:
45 67 12 34 25 39


Effective array of size 6:
45 12 34 25 39 67


Effective array of size 5:
12 34 25 39 45


Effective array of size 4:
12 25 34 39


Effective array of size 3:
12 25 34


Sorted array:
12 25 34 39 45 67
```

Note that the process stops as soon as the effective array of size 3 is found to be sorted. If the original data is almost sorted, then bubble sort can be efficient.

10.2.3   Insertion Sort

The two sorting algorithms we have looked at so far are useful when all of the data is already present in an array, and we wish to rearrange it into sorted order. However, if we are reading the data into an array one element at a time, we can take another approach — insert each element into its sorted position in the array as we read it. In this way, we can keep the array in sorted form at all times. This algorithm is called *inertion sort.*

With this idea in mind, Let us see how the algorithm would work. If the array is empty, the first element read is placed at index zero, and the array of one element is sorted. For example, if the first element read is 23, then the array is:

```
23, ?
```

We will use the symbol, ?, to indicate that the rest of the array elements contain garbage. Once the array is partially filled, each element is inserted in the correct sorted position. As each element is read, the array is traversed sequentially to find the correct index location where the new element should be placed. If the position is at the end of the partially filled array, the element is merely placed at the required location. Thus, if the next element read is 35, then the array becomes:

```
23, 35, ?
```

However, if the correct index for the element is somewhere other than at the end, all elements with equal or greater index must be moved over by one position to a higher index. Thus, suppose the next element read is 12. The correct index for this element in the current array is zero. Each element with index zero or greater in the current partial array must be moved by one to the next higher position. To shift the elements, we must first move the last element to its (unused) higher index position, then, the one next to the last, and so on. Each time we move an element we

leave a "hole" so we can move of the adjoining element, and so on. Thus, the sequence of moving elements for our example is:

```
23, 35,  ?, ?
23,  ?, 35, ?
 ?, 23, 35, ?
```

The index zero is now vacant, and the new element, 12, can be put in that position.

```
12, 23, 35, ?
```

The process repeats with each element read in until the end of input. So, if the next element is 47, we would traverse from the beginning of the array until we find larger than 47 or until we reach the end of the filled part of the array. In this case, we reach the end of the array, and insert 47:

```
12, 23, 35, 47, ?
```

Let us develop the algorithm in more detail by observing how we insert a new item, 30. The correct position is found by traversing the partial array as long as the *new item* is greater than the *array element*. In this case, the array traversal stops at index 2, since the element at index 2, namely 35, is grater than the new element, 30. In general, the following loop finds the correct position in an array, `aray`, for the new `item`. Notice we compare the index, `i`, with the variable, `freepos`, whose value is now 4, to know when we have reached the next free position in the array.

```
for (i = 0; i < freepos && item > aray[i]; i++)
      ;
```

When this loop terminates, in our case, the variable, `i`, will be 2. Next, elements from index,i(2), to index `freepos`(4) are moved over one position. The highest indexed element must be moved first, then the next highest index, and so on. The following loop moves all elements, with index greater than or equal to i, in a correct order:

```
for (k = freepos; k > i; k--)
     aray[k] = aray[k - 1];
```

When this loop terminates, the loop counter, `k`, will be equal to `i`, which is the index of the "hole" created in the array:

```
12, 23, ?, 35, 47
```

Finally, the new `item` can be inserted at index, `i`. Figure 10.12 shows the complete function for inserting one new element in a sorted array, given the array, the new item, and the next free position (which, incidentally, is the current size of the array).

The function traverses the partial array until it finds either that `item` is less than or equal to the array element or that the array is exhausted. If the array is exhausted, the second loop is not executed since `i == freepos`. In this case, the item is merely inserted at the correct position. Otherwise, elements at and above index, `i`, are moved over one position, and the new element is inserted at the correct index.

We are now ready to implement insertion sort. The program logic is simple:

```
/*    File: sortsrch.h - continued */
void insert_sorted(int aray[], int item, int freepos);


/*    File: sortsrch.c - continued */
/*    Function inserts item in sorted order in array aray. Freepos
      is the next free pos. in the array.
*/
void insert_sorted(int aray[], int item, int freepos)
{     int i, k;

      i = 0;
      /* find the correct pos. */
      for (i = 0; i < freepos && item > aray[i]; i++)
            ;

      for (k = freepos; k > i; k--)      /* move elements */
            aray[k] = aray[k - 1];
      aray[i] = item;                    /* insert new item */
}
```

Figure 10.12: Code for inserting and element

```
Repeat the following until end of input:
      read a number,
      insert the number read into the array in sorted order;
      if the array is full, break out of loop.
```

The program terminates after a printing of the sorted array. The program uses the above function, `insert_sorted()`, to insert each number in sorted order into the array, and a function, `pr_aray_line()` of Figure 10.5, to print the array. These functions are included in file, `sortsrch.c`. The program driver is shown in Figure 10.13. Notice, we increment the number of elements in the array in each call to `insert_sorted()`, since we have added a new element to the array. We have included a debug statement to print out the partial array at each step. The input is terminated either when an end of file is reached or when the array becomes full.
    Sample Session:

```
***Insertion Sort***

Type numbers to be sorted, EOF to quit
23

23
12

12 23
```

```
/*    File: insort.c
      Other Source Files: sortsrch.c
      Header Files: sortsrch.h
      Program uses input to fill a float array in sorted order.
*/

#include <stdio.h>
#define MAX 100
#define DEBUG
#include "sortsrch.h"

main()
{    int x, y[MAX],
               k;          /* no. of items in an array */

     printf("***Insertion Sort***\n\n");
     printf("Type numbers to be sorted, EOF to quit\n");
     k = 0;

     while (scanf("%d", &x) != EOF) {
          insert_sorted(y, x, k++);

          if (k == MAX) {
               printf("Array full\n");
               break;
          }

          #ifdef DEBUG
               pr_aray_line(y, k);
          #endif

     }
     printf("SORTED ARRAY\n");
     pr_aray_line(y, k);
}
```

Figure 10.13: Driver for Insertion Sort

```
35

12 23 35
30

12 23 30 35
47

12 23 30 35 47
10

10 12 23 30 35 47
^D


SORTED ARRAY
10 12 23 30 35 47
```

Insertion sort can be adapted to sorting an existing array. Each step works with a sub-array whose effective size increases from two to the size of the array. The element at the highest index in the sub-array is inserted into the current sub-array, the effective size is increased, etc. (see Problem 5).

10.3   Binary Search

As we saw earlier, the linear search algorithm is simple and convenient for small problems, but if the array is large and/or requires many repeated searches, it makes good sense to have a more efficient algorithm for searching an array. Now that we know how to sort the elements of an array, we can make use of that ordering to make our search more efficient. In this section, we will present and implement the *binary search* algorithm, a relatively simple and efficient algorithm.

The algorithm is easily explained in terms of searching a dictionary for a word. In a dictionary, words are sorted alphabetically. For simplicity, let us assume there is only one page for all words starting with each letter. Let us assume we wish to search for a word starting with some particular letter.

We open the dictionary at some midway page, let us say a page on which words start with M. If the value of our letter is M, then we have found what we are looking for and the word is on the current page. If the value of our letter is less than M, we know that the word would be found in the first half of the book, i.e. we should search for the word in the pages preceding the current page. If the value of our letter is greater than M, we should search the pages following the current page. In either case, the effective size of the dictionary to be searched is reduced to about half the original size. We repeat the process in the appropriate half, opening to somewhere in the middle of that and checking again. As the process is repeated, the effective size of the dictionary to be searched reduces by about half at each step until the word is found on a current page.

Binary search essentially follows this approach. For example, given a sorted array of items, say:

```
12, 29, 30, 32, 35, 49
```

```
/*    File: sortsrch.c - continued */
/*    Function uses binary search to search for item in the array y[]. */
int binsrch(int y[], int lim, int key)
{     int low, mid, high = lim - 1;

      low = 0;

      while (low <= high) {          /* Is the array exhausted?   */
            mid = (low + high) / 2;  /* If not, find middle index */

            if (key == y[mid])       /* Is the key here?          */
                  return(mid);       /* If so, return index.      */

            else if (key < y[mid])   /* else if key is smaller,   */
                  high = mid - 1;    /* reduce the high end;      */

            else
                  low = mid + 1;     /* otherwise, increase low   */
      }
      return(-1);                    /* Not found, return -1      */
}
```

Figure 10.14: Code for Binary Search

suppose we wish to search for the position of an element equal to x. We will search the array which begins at some low index and ends at some high index. In our case the low index of the effective array to be searched is zero and the high index is 5. We can find the approximate midway index by integer division (low + high) / 2, i.e. 2. We compare our value, x with the element at index 2. If they are equal, we have found what we were looking for; the index is 2. Otherwise, ff x is greater then the item at this index, our new effective search array has a low index value of 3 and the high index remains unchanged at 5. If x is less than the element, the new effective search array has a high index of 1 and the low index remains at zero. The process repeats until the item is found, or there are no elements in the effective search array. The terminating condition is found when the low index exceeds the high index. The algorithm is implemented as a function in Figure 10.14.

We use the binsrch() function in an example program which repeatedly searches for numbers input by the user. For each number, it either gives the index where it is found or prints a message if it is not found. An array in sorted form is initialized in the declaration. The code for this driver is shown in Figure 10.15

Sample Session:

```
***Binary Search***

The array is:
12 29 30 32 35 49
```

```
/*    File: bsrcharay.c
      Other Source Files: sortsrch.c
      Header Files: sortsrch.h
      Program uses binary search to search a sorted array of numbers.
*/

#include <stdio.h>
#define MAX 100
#define DEBUG
#include "sortsrch.h"

main()
{     int i, x, y[MAX] = {12, 29, 30, 32, 35, 49};
      int k = 6;              /* no. of items in the array y[] */

      printf("***Binary Search***\n\n");
      printf("The array is:\n");
      pr_aray_line(y, k);
      printf("Type a number, EOF to quit: ");

      while (scanf("%d", &x) != EOF) {
          i = binsrch(y, k, x);

          if (i >= 0)
              printf("%d found at array index %d\n", x, i);
          else
              printf("%d not found in array\n", x);

          printf("Type a number, EOF to quit: ");
      }
}
```

Figure 10.15: Test Driver for Binary Search

```
Type a number, EOF to quit:   12
12 found at array index 0

Type a number, EOF to quit:   23
23 not found in array

Type a number, EOF to quit:   34
34 not found in array

Type a number, EOF to quit:   45
45 not found in array

Type a number, EOF to quit:   30
30 found at array index 2

Type a number, EOF to quit:   29
29 found at array index 1

Type a number, EOF to quit:   ^D
```

10.4   An Example — Payroll Data Records

So far, in the previous sections, we have seen how to search and sort an array of integers. In this section we apply the sort and search methods to our database of payroll records. The data items in a payroll record are id number, hours worked, rate of pay, regular and overtime pay. Our task is to write an interactive program which displays the pay record for a given individual.

We saw how we could implement such a database in Chapter 7. There, the data record for a specific id is stored at the same index in several different arrays as shown in Figure 10.16. In our application, we will search the database to find the payroll record given a specific id number as the key. Therefore, we will need to sort the database by the id number field. When we search for the key, we will get the index for the element, if any, which matches the sought after id. With that index in id number array, we can access the remaining information for that data record.

As we saw in the previous section, when we sort an array, we rearrange the positions of the array elements. When we sort data records, we must rearrange the positions of all fields of the data records; i.e. if a data record is spread over several arrays, we must rearrange the elements of all of these arrays in an identical manner. In this way, we will still be able to access a data record using the index determined by a key. To sort the database, we can use either selection sort or bubble sort for our task. We will assume that the input data is mostly sorted, requiring little rearrangement, and therefore will choose to use bubble sort. This is a reasonable assumption since records are usually kept in a file in sorted order. Only new records entered may be out of place. We will modify `bubblesort()` of the last section to handle data records.

The input data record is spread over three arrays; namely, `id[]`, `hrs[]`, and `rate[]`. Since we are sorting records by id numbers, the decision whether to swap records is determined by the elements of the `id[]` array; however, if we swap elements of `id[]`, we must also swap corresponding

	id	hrs	rate	regular	overtime
index 0					
index 1					
index i	5	20.0	10.0	200.0	0.0
index MAX-1					

Figure 10.16: A Data Record Across Arrays

elements of the other two arrays.

The code for the modified sort function, called `sortdata()`, that sorts input payroll records is shown in Figure 10.17. We write the code in the file `payutil.c` and add its prototype in `payutil.h`. These files have other payroll functions and prototypes developed in Chapter 7, including: `getdata()` which reads the input data, `calcpay()` which calculates regular and overtime pay, and `printdata()` which prints the pay records in a table. We also use the file `tfdef.h` that defines TRUE and FALSE.

We can now use the above function in a payroll program that sorts the input data before processing it. The main purpose of this program is to test the operation of creating a sorted database of records before later modifications to the program. The driver is very simple and consists of functions that get data, sort data, calculate pay, and print data as seen in Figure 10.18. Notice we have performed the calculate pay step after the database has been sorted, as the arrays containing this data are not rearranged by our sort function. The sample session is shown below.

Sample Session:

```
***Payroll Program - Sorted Data***

ID <zero to quit>:  2
Hours Worked:  20
Rate of Pay:  5

ID <zero to quit>:  5
Hours Worked:  40
Rate of Pay:  10

ID <zero to quit>:  12
Hours Worked:  45
Rate of Pay:  12.50
```

```
/* File: payutil.c - continued */
#include "tfdef.h"
void sortdata(int id[], float hrs[], float rate[], int lim)
{    int i, k, temp, swap = TRUE;
     float ftmp;

     for (i = 0; swap && i < lim - 1; i++) {
          swap = FALSE;

          for (k = 0; k < lim - i - 1; k++)
               if (id[k] > id[k + 1]) {
                    temp = id[k];
                    id[k] = id[k + 1];
                    id[k + 1] = temp;

                    ftmp = hrs[k];
                    hrs[k] = hrs[k + 1];
                    hrs[k + 1] = ftmp;

                    ftmp = rate[k];
                    rate[k] = rate[k + 1];
                    rate[k + 1] = ftmp;
                    swap = TRUE;
               }
     }
}

/* File: payutil.h - continued */
void sortdata(int id[], float hrs[], float rate[], int lim);
```

Figure 10.17: Code for `sortdata()` and header file entry

```
/*     File: paysrt.c
       Other Source Files: payutil.c
       Header Files: payutil.h
       Program calculates payroll data for a number of id's. It
       gets data, sorts data, calculates pay, and prints data for
       all id's.
*/

#include <stdio.h>
#include "payutil.h"
#define MAX 10

main()
{      int i, n = 0, key, id[MAX];
       float hrs[MAX], rate[MAX], regpay[MAX], overpay[MAX];

       printf("***Payroll Program - Sorted Data***\n\n");
       n = getdata(id, hrs, rate, MAX);
       sortdata(id, hrs, rate, n);
       calcpay(hrs, rate, regpay, overpay, n);
       printdata(id, hrs, rate, regpay, overpay, n);
}
```

Figure 10.18: Test driver for `sortdata()`

```
ID <zero to quit>:   8
Hours Worked:   50
Rate of Pay:   14


ID <zero to quit>:   0


***PAYROLL: FINAL REPORT***
 ID    HRS    RATE      REG     OVER      TOT

  2  20.00    5.00   100.00     0.00   100.00
  5  40.00   10.00   400.00     0.00   400.00
  8  50.00   14.00   560.00   210.00   770.00
 12  45.00   12.50   500.00    93.75   593.75
```

The program file, `paysrt.c`, containing the driver, and the source file, `payutil.c`, with the new function, `sortdata()` added, must be compiled and linked. Observe that the input is almost sorted by id number; only the last record is out of place. Bubble sort can sort this data in one pass.

Having observed that the database is correctly sorted, we can now complete the task to search for a specific record to display. We will use binary search on the data in sorted order. The search for an id number in the array, `id[]`, returns an index if it is found, and returns -1 otherwise. If the index is non-negative, the same index is used to access the rest of the data record spread over the other arrays. We modify the above test program to read the data records and calculate the pay, then repeatedly call `binsrch()` to return the index of a data record for a specified id number. If a data record exists, it is printed by `printrec()`. We have already implemented the function `binsrch()` and included it in the file, `sortsrch.c`. We will soon write `printrec()`. The program that implements our task is shown in Figure 10.19.

The first part of the program reads data, sorts data, and calculates pay. The second part of the program reads an id number and calls `binsrch()` to locate its index in the array. If the index is non-negative, the program uses a function, `printrec()`, to print a data record at that index. If the index is negative, the program prints an error message. The function, `printrec()`, is shown in Figure 10.20 and added to the file, `payutil.c`.

A sample session for the search part of the program is shown below. The input data is assumed identical to that in the sample session for the previous program `paysrt.c`.

```
    ***Payroll Program - Search Data***

 Type an id <zero to quit>:   8
 ***PAYROLL RECORD FOR ID 8***

  ID    HRS    RATE      REG     OVER      TOT
   8  50.00   14.00   560.00   210.00   770.00

 Type an id <zero to quit>:   10
 Error - no such id


 Type an id <zero to quit>:   2
 ***PAYROLL RECORD FOR ID 2***
```

```
/*    File: paysrch.c
      Other Source Files: payutil.c, sortsrch.c
      Header Files: payutil.h, sortsrch.h
      Program sorts and calculates payroll data for a number of id's. It
      then uses sequential search to find and print data records for
      specified id numbers.
*/

#include <stdio.h>
#include "payutil.h"
#include "sortsrch.h"
#define MAX 10

main()
{     int i, n = 0, key, id[MAX];
      float hrs[MAX], rate[MAX], regpay[MAX], overpay[MAX];

      printf("***Payroll Program - Search Data***\n\n");
      n = getdata(id, hrs, rate, MAX);
      sortdata(id, hrs, rate, n);
      calcpay(hrs, rate, regpay, overpay, n);
      printdata(id, hrs, rate, regpay, overpay, n);

      printf("Type an id <zero to quit>: ");
      while (scanf("%d", &key) != EOF && key != 0) {
            i = binsrch(id, n, key);

            if (i >= 0)
                  printrec(id, hrs, rate, regpay, overpay, i);
            else printf("Error - no such id\n");

            printf("Type an id <zero to quit>: ");
      }
}
```

Figure 10.19: Code for searching the database

```
/* File: payutil.c - continued */
/* Function prints a single data record at a specified index. */
void printrec(int id[], float hrs[], float rate[],
              float reg[], float over[], int i)
{
    printf("***PAYROLL RECORD FOR ID %d***\n\n", id[i]);
    printf("%10s%10s%10s%10s%10s%10s\n", "ID", "HRS",
           "RATE", "REG", "OVER", "TOT");

    printf("%10d%10.2f%10.2f%10.2f%10.2f%10.2f\n",
           id[i], hrs[i], rate[i], reg[i], over[i],
           reg[i] + over[i]);
}


/* File: payutil.h - continued */
void printrec(int id[], float hrs[], float rate[],
              float reg[], float over[], int i);
```

Figure 10.20: Code for `printrec()` and header file entry

```
    ID   HRS  RATE     REG  OVER     TOT
     2 20.00  5.00  100.00  0.00  100.00
Type an id <zero to quit>:  0
```

10.5   Polymorphic Data Type

Very often in programs, a generic operation must be performed on data of different types. For
example, in our bubble sort algorithm for the payroll records, when elements were found out of
order in the `id[]` array, we needed to swap the integer elements in that array as well as the float
elements in the `hrs[]` and `rate[]` arrays. If we decided to implement this swapping operation
as a function, we would need to write two functions: one to swap integers, and another to swap
floating point values; even though the algorithm for swapping is the same in both cases. (We
wrote a swap function for integers using pointers in Chapter 6).

The C language provides a mechanism which allows us to write a single swapping function
which can be used on any data type. This mechanism is called a *polymorphic data type*, i.e. a data
type which can be transformed to any distinct data type as required. An item of polymorphic
data type is created by the use of a *generic pointer*. A generic pointer is simply a byte address
without an associated type. In other words, a generic pointer does not point to an object of a
specific type; it just points to some location in the memory of the computer. In ANSI C, a generic
pointer is declared as a void pointer (in old C, a generic pointer is a char pointer). It is only when
the actual operations must be performed on the data that generic pointers are cast to pointers to
specific types and dereferenced.

Using the concept of a generic pointer, we can assume the following prototype for a function

```
/* File: payutil.c - modified */
#include "tfdef.h"
void sortdata(int id[], float hrs[], float rate[], int lim)
{    int i, k, temp, swap = TRUE;
     float ftmp;

     for (i = 0; swap && i < lim - 1; i++) {
         swap = FALSE;

         for (k = 0; k < lim - i - 1; k++)
             if (id[k] > id[k + 1]) {
                 gen_swap((void *)(id + k), (void *)(id + k + 1), 'd');
                 gen_swap((void *)(hrs + k), (void *)(hrs + k + 1), 'f');
                 gen_swap((void *)(rate + k), (void *)(rate + k + 1), 'f');

                 swap = TRUE;
             }
     }
}
```

Figure 10.21: Modified code for `sortdata()` using generic swap

to swap two data items of any type:

```
void gen_swap(void * x, void * y, char type);
```

Here, `x` and `y` are generic pointers to two data items, and `type` specifies the type of the data using a single character. With this information, we can now rewrite the function, `sortdata()`, in the file, `payutil.c` using `gen_swap` to swap all data items. The code is shown in Figure 10.21. Notice in the calls to `gen_swap()` we cast the pointers to the integer array elements (`id + k` and `id + k + 1`) to void pointers. Similarly, the pointers to the float data items in the `hrs[]` and `rate[]` arrays are cast to void pointers. We pass the character constants `'d'` for integer, or `'f'` for float to tell `gen_swap()` the type of the data it is to swap.

We can now write the code for `gen_swap()` as seen in Figure 10.22. We have declared two temporary variables, `temp` and `ftmp` to hold an integer or float value, respectively when we do the swapping. The variable, `type` is used to switch to the appropriate code sequence to swap the two data items.

If we made these modifications to `payutil.c` and recompiled our program, it would behave exactly as it did in the last section. Of course, as we stated earlier, we may not want to use a function to perform the swap in bubble sort because of the overhead in calling and returning from a function. As another example of the use of the polymorphic data type, consider writing a function that will print an array, regardless of type, with five elements per line. We may have an array of integers and an array of floats to be printed and wish to use the same function to format the lines of output. Figure 10.23 shows a driver program and the function, `praray()`.

The function calls to `praray()` pass the array pointers after first casting them to generic pointer types. In the function, `praray()`, we use the array index to determine when a new line is

```
/* File: payutil.c - continued */
void gen_swap(void * x, void * y, char type)
{  int temp;
   float ftmp;

   switch(type)
   { case 'd' : temp = *(int *)x;
                *(int *)x = *(int *)y;
                *(int *)y = temp;
                return;
     case 'f' : ftmp = *(float *)x;
                *(float *)x = *(float *)y;
                *(float *)y = ftmp;
                return;
     default  : printf("Error in gen_swap: %c not a legal type\n",type);
   }
}
```

Figure 10.22: Code for gen_swap

needed. Since we wish to print five elements to a line, a newline is printed every time the index, i, is a multiple of 6, i.e. i % 6 is zero. When the index i is zero, no newline is needed.

The void pointer, y, points to the array and the type value is a character, 'd' for integers, and 'f' for float, as before. Each element of the array is printed by means of a switch statement. The switch cases are selected by the type of the array passed. If the type is 'd', a decimal integer is printed; if the type is 'f', a float is printed. If desired, the function can be extended to handle other types as well. Let us examine the printing of an ith element of an integer array. For a type 'd', the argument expression in printf() is:

```
*((int *) y + i)
```

The void pointer, y, is first cast to the desired type, i.e. int *; then, the int * is increased by i so as to point to the ith element of an integer array. This pointer is finally dereferenced to access the ith element of the array. Thus, printf() prints the value of the ith element of an integer array. Similarly, a float array element is printed out by first casting the generic pointer to a float pointer. A sample output is shown below.

```
***Generic Pointers and Polymorphic Data Types***

Integer array is:
12 23 34 45 72

Float array is:
12.240000 23.350000 43.570000 82.209999
```

Use of polymorphic data types makes for compact programs; however, their use is not recommended for beginning programmers. For the most part, we will not use them in this text.

```
/*    File: genptr.c
      Program shows the use of generic pointers to implement a
      polymorphic data type. An integer and a float array are printed
      out by the same primitive function praray().
*/
#include <stdio.h>
void praray(void * y, int lim, char type);


main()
{    int x[] = {12, 23, 34 ,45, 72};
     float y[] = {12.24, 23.35, 43.57, 82.21};

     printf("***Generic Pointers and Polymorphic Data Types***\n\n");
     printf("Integer array is:\n");
     praray((void *) x, 5, 'd');
     printf("Float array is:\n");
     praray((void *) y, 4, 'f');
}


/*    Function prints an array of any type, int or float. */
void praray(void * y, int lim, char type)
{    int i;

     for (i = 0; i < lim; i++) {
          if (i != 0 && i % 6 == 0)      /* add a newline every 6th item */
               printf("\n");

          switch(type) {
               case 'd': printf("%d ", *((int *) y + i));
                         break;
               case 'f': printf("%f ", *((float *) y + i));
                         break;
               default: printf("Error in printing array\n");
          }

     }
     printf("\n");
}
```

Figure 10.23: Code for printing arbitrary arrays

10.6 Common Errors

1. In insertion sort, the elements are shifted incorrectly. Shift the highest index element first, then the next highest, and so forth.

2. The argument in binary search that specifies the high index is incorrect. If the size of the array is passed as the highest index, there is a problem. If the size of the array is n, the index n is outside the array. The argument should be n - 1.

3. Generic pointers should be used with care. In traditional C, use char pointer instead of void pointer.

10.7 Summary

In this chapter we have developed algorithms for searching a collection of data for a specific element, called the *key*. We saw a simple algorithm, *linear search* which started at the beginning of the data, and compared each element against the key until it was found, or the data was exhausted. However, linear search is not very efficient if the number of elements to search is large. In order to develop more efficient algorithms, we need to take advantage of the order of the data. Therefore, we next discussed how we can arrange the elements in an array in a specified order — a process called *sorting*. We developed three sorting algorithms: *selection sort*, *bubble sort*, and *insertion sort*. The first two of these are useful when all of the data is already stored in an array, and insertion sort can be used to sort the data as it is being read into the array.

Once we have the data sorted, we developed a more efficient searching algorithm — *binary search*. This algorithm worked by dividing the data in half, and deciding in which half the key would occur. With each step, then, we can eliminate half of the data from further consideration.

These searching and sorting techniques are general and may be applied to any type of data. We used them in our payroll task to find individual payroll records in a database given an id as the key.

Finally, we discussed the use of the polymorphic data type, or *generic pointers* to implement a common operation that may be applied to data of different types.

10.8   Exercises

Find and correct errors if any.

1. ```
   main()
   {     int x[10];

         x[10] = {12, 23, 45};
   }
   ```

2. ```
   main()
   {     int x[10];

         x = {12, 23, 45};
   }
   ```

3. ```
   main()
   {     int i, x[10];

         for (i = 0; i < 10; i++)
               x = 0;
   }
   ```

4. Should you use a function or a macro to swap values in bubble sort? Explain your reasons.

5. Bubble sort moves the largest value to the highest index. Modify the bubble sort code to move the smallest element to the lowest index.

6. Insertion sort inserts a new element into the array. Modify the insertion sort method to apply it to an unsorted array with n elements. Do not use another array.

7. Modify the bubble sort to apply it to an array of characters housing a string. The number of elements in the string are unknown, but terminated by a NULL.

10.9 Problems

1. Write a function that sorts an array of integers in decreasing order.

2. Write a function that sorts an array of integers in either increasing or in decreasing order as specified by an argument.

3. Write a binary search function that searches an array of integers sorted in decreasing order.

4. Write a binary search function that searches an array of integers sorted in either increasing or in decreasing order as specified by an argument.

5. Write a function that uses insertion sort to sort an array of input numbers, either in increasing or in decreasing order.

6. Write a function that uses insertion sort to sort an existing array of integers.

7. Write a program to read an array of integers from a file. Write a function that takes two arguments: low and high. Low and high specify the low and high indices of an effective array. Function finds indices for the maximum and the minimum elements in the specified effective array. Use the function with zero for low and the highest valid index for high. Print the values of maximum and minimum.

8. Repeat the last problem, but this time swap the largest element in the effective array with the one at the high index, say index n; swap the smallest element with the one at the low index.

9. Repeat the last problem, but this time after the swap of the elements change the effective array so low is 1 and high is n - 1. Repeat the process so the largest and smallest elements are found in the array from index 1 through n - 1. Swap the next largest element with the one at index n - 1, and swap the next smallest with the one at index 1. The next swap considers the array from index 2 through n - 2, etc until all elements of the array are in increasing order. This is another way of sorting an array.

10. Compare the operations involved in the above sorting with that for bubble sort. What are the approximate comparisons required to sort an array of n items by the two methods.

11. Write a menu_driven program that allows the following commands: get data from a file, add data, delete data, sort data, search data, save data to a file, help, quit. Assume that data records consist of id numbers and exam scores. Sorting must be done either by id numbers or by exam scores and it must be either in ascending or descending order.

12. Repeat the last problem, but get data uses insertion sort to read data in sorted form by id numbers.

13. Write a program that reads integers into an array A. Use another array P of the same size to store each index of the array A in the following way. The index in A with the smallest element is stored at index 0 of P, the index of the next smallest element in A is stored at index 1 of P, and so on. Print the array A, and print the elements of A ordered in the sequence given by each succeeding index stored in P.

14. Repeat Problem 11, but use the approach of Problem 13 to sort the data.

15. Repeat Problem 11, but use insertion sort to read data in sorted form by id numbers. The sort command then uses approach of Problem 13 to sort the data by exam scores.

16. Compare the operations required for sorting in Problems 11 and 14.

17. Write a program to merge two sorted arrays A and B into a third array S as follows. Start with initial index, ia, of the element to be merged from A and also the index, ib, of the element from B. If the element from A is smaller than the element from B, append that element of A to the array S and increment ia; if the element of B is smaller than that of A, add the element from B to S and increment ib. If they are equal, add both elements to S and increment both ia, ib. If either array is exhausted, copy the elments from the other array into S. Repeat until both arrays are exhausted. Print A, B, and S.

18. Develop a sort method using the merging of two arrays as in the last problem. Given an array with 8 elements, assume it is split into as many arrays as there are elements. Merge each adjacent pair into 4 arrays of two elements each. Merge each pair again into 2 arrays of 4 elements each. Finally, merge the two into a sorted array. The method can be applied to any array and is called merge sort method. Write a program that sorts an array by merge sort.

19. Write a program that sorts the characters in a string.

20. Write a program that reads strings; for each string compute the frequency of occurrence of each character.

21. Write a program that reads text from a file and computes cumulative frequency of occurrence of each character.

22. Write a program that searches a string for a specified character and returns the index of its first occurrence.

23. Write a program that returns the first occurrence of a character in a string starting at some specified index.

24. Write a program to find all occurrences of a character in a string.

25. Write a program that replaces all the occurrences of a character in a string by another character.

26. Write a program that sorts characters in a string according to a different order than that of the ASCII values. Use a function to compare two characters and return whether one is greater than, equal to, or less than the other. The function first converts all lower case letters to upper case and then compares their ASCII values. The program sorts characters ignoring case.

27. Write a sort program similar to the above problem for integers. Use a function that compares the absolute values of two integers. The program sorts by absolute values.

28. Write a character sort program that uses a function, `cmpchrs()`, to compare two characters by an arbitrary criteria. Assume that an array stores all ASCII printable characters in an assumed increasing order: vowels first, consonants next, digits next, all others next. The function, `cmpchrs()`, looks up the corresponding indices of characters to compare characters. The result of comparing the indices is returned by the function. The program sorts characters in an order specified by `cmpchrs()`.

Chapter 11

String Processing

These days, computers are not used only for processing numbers, but, as we have seen in previous chapters, they are also used for processing textual data. As we saw in Chapter 7, in C, textual data is represented using arrays of characters called a *string*. If we are to manipulate strings in any reasonable way we must have several basic operations available to us. Since string is not a basic type in C, string operations must be developed as functions. A library of such functions can then be used as a part of the language. In other words, we can effectively treat string as an *abstract data type* [1] once we have string operations written in function form. The Standard Library provides a rich set of functions for performing operations on strings such as copying and comparing them, breaking strings up into parts, joining them, finding substrings, substituting one string for another, and so forth. In this chapter, we will discuss such string processing using the built-in library functions as well as look at how some of these functions can be written.

We begin the chapter by defining a user defined data type for strings and then use this new type throughout. We then describe the library functions available for performing string operations, including reading and writing strings, copying a string and finding its length, comparing and joining strings and converting the information in a string to other data types. We conclude the chapter with several example programs using these string operations.

11.1   The Data Type `STRING`

Because a string is such a common data structure in programs, it may be convenient to define it as its own data type. We can then define functions to perform operations on operands of the defined data type; effectively treating the defined type as a new data type (an abstract data type). As we have seen previously, a string is implemented as an array of characters, and an array is implemented as a contiguous collection of cells and a *pointer* pointing to the beginning of the block. The name of the array is associated with this pointer cell, rather than with the data cells themselves. When we pass an array to a function, we pass this pointer to the array. So when we are processing strings, passing them to functions, returning them as values, we are handling pointer values. Therefore, we can define a data type, `STRING`, as a pointer to a character as follows:

```
typedef char * STRING;
```

---

[1]To define an abstract data type, we must define a way to declare variables of that type together with operations that can be performed on such data items. A full description of the concept of abstract data types is beyond the scope of this text; however, the basic idea is presented here.

We can then define string variables in terms of the data type, `STRING`:

```
STRING s, t;
```

The variables, `s` and `t`, are character pointers. We can access the characters in the string by dereferencing the pointer or using array type indexing. But remember, declaring a pointer type only allocates space for the pointer; it does not allocate cells for an array; it does not initialize the pointer. We cannot use a `STRING` variable to store a string of characters, but merely to point to a pre-allocated string. We must always declare a character array to store a string of characters, and can then initialize a `STRING` variable to point to this array. `STRING` does not serve to allocate memory for a string. As such, the concept of abstract data type is not totally satisfied by the above type definition; however, with the above caveat, we can otherwise treat it as such.

We illustrate the use of the `STRING` type by writing a rogram to read and print a string as shown in Figure 11.1. Note, in `main()`, a character array, `s`, is declared. The name of the array, `s`, is an (initialized) character pointer — the same as our type, `STRING`, and may therefore be passed to the function `our_strprint()` which expects a `STRING` argument. Notice, we have placed the `typedef` for `STRING` in a header file, `strtype.h` since it will be useful for other programs we will write.

11.2   Library String Functions

With a data type defined, we may now proceed to define functions to implement the operations on data of this type. As mentioned above, the C built-in library provides a rich set of string processing functions. We describe some of the more common ones here; others are described in Appendix C.

11.2.1   String I/O: *gets()* and *puts()*

One of the first operations we may need for strings is the ability to read or write strings from the standard input or to the standard output. For example, if we had a task:

STR0: Read strings until end of file, convert each string to upper case, and print the modified string.

we could easily write an algorithm for the task:

```
while not EOF, read a string
     convert string to upper case
     print string
```

To implement this algorithm, we need three functions: read a string from standard input stream, write a string to standard output, and convert a string to upper case. We have shown crude versions of these first two functions in the previous, section; however, the standard library provides these operations as well. Library function, `gets(s)` reads a string from the standard input into an array pointed to by `s`; and, `puts(s)` writes a string pointed to by `s` to the standard output. The prototypes for these functions, declared in `stdio.h`, are: .

```
STRING  gets(STRING  string);
int puts(STRING  string);
```

```
/*  File: strtype.c
    This program illustrates the use of a type definition for strings.
*/
#include <stdio.h>
#include "strtype.h"
#define SIZE 100
void our_strprint(STRING s);
void our_strread(STRING s);
main()
{   char s[SIZE];        /* allocate space for a string */

    our_strread(s);            /* read a string */
    our_strprint(s);    /* print a string */
}


/*  Function reads a string from standard input.*/
void our_strread(STRING s)  /* declare a STRING type */
{
    while ((*s = getchar()) != '\n')
        s++;
    *s = NULL;
}


/*  Function writes a string to standard output.*/
void our_strprint(STRING s)  /* declare a STRING type */
{
    while (*s) {
        printf("%c", *s);
        s++;
    }
    printf("\n");
}


/*  File: strtype.h
    This file contains the definition of type STRING
*/


typedef char * STRING;
```

Figure 11.1: Program illustrating the STRING data type

If reading is successful, `gets()` returns the pointer to the string; otherwise, it returns a `NULL` pointer, i.e. a pointer whose value is zero. A returned value of `NULL` usually implies an end of file. When `gets()` reads a string, it reads the input characters until a newline is read, discards the newline, appends a `NULL` character to the string, and stores the string where `s` points. Similarly, `puts()` outputs the string, `s`, after stripping the `NULL` and appending a newline. It returns the last character value output if successful; otherwise it returns `EOF`. Note, the arguments to these functions and the return value from `gets()` are character pointers, i.e. equivalent to our `STRING` data type, and we can consider them as such. The argument of `gets()` **MUST** be a string; otherwise, the function attempts to store characters wherever the argument points, which can create a possibly fatal error when the program executes.

We will write and use a function, `ucstr()`, which converts a string to upper case. The whole program is simple: it reads a string, converts it to upper case, and prints it; and is shown in Figure 11.2 In the driver, the loop expression reads a line into `s`; if successful, the returned value is a non-zero pointer, `s`, and the loop is executed. In the loop body, the string, `s`, is converted to upper case, and printed. The function, `ucstr()`, converts a string to upper case by traversing the string and converting each character to upper case using library routine, `toupper()`, which returns the upper case version of its argument if it is a lower case letter; otherwise it returns the argument unchanged.

Sample Session:

```
***String to Upper Case***

Type strings, EOF to terminate
Hello
HELLO
Pad 19A
PAD 19A
good morning
GOOD MORNING
^D
```

The above program reads lines until end of file. As a slight variation on this task, sometimes it is desirable to loop until a blank line is entered. Here is a loop that copies lines until a blank line is entered:

```
while (*gets(s))
    puts(s);
```

Assuming that a line is read successfully, `gets()` returns `s`. The expression, `*gets()`, is the same as `*s`, which is the first character in the string, `s`. As long as the first character of `s` has a non-zero value, the loop continues. When the first character is a `NULL`, the loop terminates. If a blank line is entered by typing a *RETURN*, `gets()` reads an empty string and the loop terminates.

We can also use `gets()` in a menu driven program which requires the user to enter either a single character or a command line. In our previous menu driven programs in Chapter 4, we saw that reading a single command character required that the keyboard buffer be flushed of the newline character before reading the next command. If only one character is to be read, or if the first character of a command line is sufficient to identify a command, then it is simpler to read the

```
/*  FIle: ucstr.c
    This program reads strings, converts them to upper case, and
    prints them out.
*/

#include <stdio.h>
#include <ctype.h>                   /* includes toupper() */
#include "strtype.h"
#define SIZE 100
void ucstr(STRING t);

main()
{
    char s[SIZE];       /* allocate a string  */

    printf("***String to Upper Case***\n\n");
    printf("Type strings, EOF to terminate\n");

    while (gets(s)) {
        ucstr(s);
        puts(s);
    }
}

/* Converts t to upper case string */
void ucstr(STRING t)
{

    while (*t) {                     /* loop until char is null */
        *t = toupper(*t);            /* convert char *t to upper case */
        t++;                         /* point to next char */
    }

}
```

Figure 11.2: Program to read and print strings using `gets()` and `puts()`

entire line using `gets()`, which strips the newline character from the input line, and then examine only the first character of the input string. Here is a loop for a menu driven program driver:

```
printf("H(elp, Q(uit, D(isplay\n");
while (gets(s)) {
    switch (toupper(*s)) {

        case 'H': help();
                  break;

        case 'Q': exit(0);

        case 'D': display();
                  break;

        default: ;
    }

    printf("H(elp, Q(uit, D(isplay\n");
}
```

The loop reads an input string, `s`, and passes the first character of `s`, `*s` to `toupper()` which converts it to upper case. One of the cases in the switch is selected and an appropriate function is executed. The loop repeats until `gets()` returns end of file.

We may now use library functions, `gets()` and `puts()`, in place of functions we have previously written ourselves to read and write strings. Remember, `gets()` reads an entire line of input text into a string; replacing the newline with a `NULL`. Likewise, `puts()` prints an entire `NULL` terminated string; adding a newline at the end.

11.2.2   String Manipulation: *strlen()* and *strcpy()*

As our next task, let us consider reading lines of text and finding the longest line in the input:

STRSAVE: Read text lines until end of file; save the longest line and print it.

Our approach is similar to the algorithm for finding the largest integer in a list of integers. We save the current "guess" at the longest line in a string, and, as each new line is read, we compare the length of the new line with that of the current longest line. If the length of the new line is greater than that of the current longest, we will save the new line into the longest and proceed. To begin, we initialize the longest line to an empty string; the shortest of all strings. Here is the algorithm:

```
    initialize longest to an empty string

    while not EOF, read a line
        if length of new line > length of current longest
            save new line into longest

    print longest
```

To implement this algorithm, we must consider how we can perform the required operations on the strings holding the new line and the current longest line. We already know how to read and write strings; we also need the operations of finding the length of a string and saving a string. For the former task, the standard library provides a function:

```
int strlen(STRING s);
```

which returns the length of a string,s, i.e. the number of characters in s excluding the terminating NULL.

For the second operation, we can consider the implementation of the maximum integer algorithm and how we saved the new maximum value — we used an assignment operator. However, this will not work for strings. Remember, the string is implemented as a character pointer. If we simply assigned one string variable to another, we would only be saving the pointer to the first string, not the string characters themselves. Then, when we read the next input line, we would overwrite the current string as well. Instead we need to *copy* the new line string into the current longest string. The standard library provides a function for this operation:

```
STRING strcpy(STRING dest, STRING source);
```

which copies a string pointed to by source into a location pointed to by dest. The function returns the destination pointer, dest. This is the equivalent of an assignment operation for data type, STRING.

The prototypes for these and other standard library string functions are in a header file, string.h. We can now write the program implementing our algorithm as shown in Figure 11.3. Notice, we initialize the current longest string by using strcpy() to copy an empty string into longest. It is also possible to initialize it as follows:

```
*longest = '\0';
```

or,

```
longest[0] = '\0';
```

Use of strcpy() makes it clear that an empty string is copied into longest. It has the *flavor* of assigning a string constant to another string, the same way longest is updated to the new string, s, within the loop body. Thus, we are sticking with our concept of an abstract data type by only using the defined functions to perform operations on data of the type, STRING. A sample session is shown below:

```
***Longest Line***

Type text lines, empty line to quit
hello
good morning

Longest line is:
good morning
```

Remember that assignments **cannot** be used to store strings into arrays. When a string is to be stored into a specified character array, use strcpy() to copy one string to another; **do NOT** use an assignment operator.

```
/*  File: long.c
    This program reads lines of text and saves the longest line.
*/
#include <stdio.h>
#include <string.h>
#define SIZE 100
#define DEBUG

main()
{   char s[SIZE], longest[SIZE];

    printf("***Longest Line***\n\n");
    strcpy(longest, "");  /* length of empty string is zero */
    printf("Type text lines, empty line to quit\n");

    while (*gets(s))
        if (strlen(s) > strlen(longest))
            strcpy(longest, s);
    printf("Longest line is: \n");
    puts(longest);
}
```

Figure 11.3: Program to find the longest string

Implementing `strcpy()`

The standard library provides the function `srtcpy()` for us to use; however, it is instructive to look at how such a function can be written. Let us write our version of `strcpy()` to copy string, `t`, into string, `s`:

```
/* File: str.c */
/* Function copies t into s */

#include "strtype.h"

STRING our_strcpy(STRING s, STRING t)
{
    while (*t != '\0') {
        *s = *t;
        s++;
        t++;
    }
    *s = '\0';
    return s;
}
```

The arguments passed to formal parameters, `s` and `t`, are of type `STRING`, i.e. character pointers. The loop is executed as long as `*t` is not `NULL`. In each iteration, a character is copied into (the string pointed to by) `s` from (the string pointed to by) `t` by the assignment of `*t` to `*s`. The pointers `s` and `t` are then incremented so they point to the next character positions in the two arrays. If `t` does not point to a `NULL`, the loop repeats and copies the next character, etc. If `t` points to a `NULL`, the loop terminates. After the loop terminates, a terminating `NULL` is appended to `s`. The function returns the pointer, `s`.

Notice, there is a problem with this implementation. The function returns the value of `s`, however, this is no longer a pointer to the destination string — `s` has been incremented as the string was copied and now points to the end of the destination string. We leave the repair of this function as an exercise (see Problem 11).

Several alternate versions of `our_strcpy()` can be written as follows (Note: these versions return `void` rather than a `STRING`):

```
/* File: str.c - continued */
void our_strcpy2(STRING s, STRING t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

In the above, the `while` condition uses the assignment expression whose value is the character assigned to check against `NULL`. If the value is `NULL`, the loop is terminated; however, the assignment places the terminating `NULL` character before the loop is terminated. Here is another variation:

```
/* File: str.c - continued */
void our_strcpy3(STRING s, STRING t)
{
    while (*s = *t) {
        s++;
        t++;
    }
}
```

In the while loop, when the assigned character is '\0', the value of the expression is zero, and
therefore false. Otherwise, the character assigned is not NULL, and the value of the expression is
true. The loop terminates correctly when it should. It is also possible to include increments in
the while expression:

```
while (*s++ = *t++)
    ;
```

Here, *t is assigned to *s, and then s and t are incremented. The next version uses array indexing;
otherwise, it is identical to the last version:

```
/* File: str.c - continued */
void our_strcpy4(STRING s, STRING t)
{ int i;

    i = 0;
    while (s[i] = t[i]) {
        i++;
    }
}
```

Memory Allocation for Strings

When a function is used to put values into an array, it is important that memory for the array be
allocated by the *calling* function. Consider the following possible error:

```
/* COMMON BUG */
    char *s;                    /* should be: char s[SIZE]; */

    strcpy(s, "Hello, good morning to all");
```

The pointer variable, s, can store only a pointer value; no memory is allocated for a string of
characters. Nor is the pointer variable s initialized. The function, strcpy(), assumes that s
points to memory where a string can be stored. No such memory has been allocated, nor does s
point to any valid location — the program will crash.

A second type of error can occur if the calling function does not allocate memory for a string,
but instead depends on the called function to do so. Let us consider an example in which a string
copy function allocates memory for the copied string and returns a pointer to it, and see where
the error leads us. Here is the function:

```
/* File: allocerr.c */
#include <stdio.h>
#include "strtype.h"

/* COMMON ERROR */
STRING scopy(STRING t)
{    char s[100];
     int i = 0;

     while (s[i] = t[i])
            i++;

     return s;
}
```

The function copies a string into an (automatic) array variable defined in the function, and returns a pointer to the array. When the function returns to the calling function, the memory for the array, `s`, is freed automatically. The value of `s` is returned, but `s` now points to garbage. Of course, the compiler does not flag an error, since the value of `s` can be legitimately returned. The fact that it now points to garbage is a program logic error.

Let us see what happens when we use this function in a program. We declare a STRING variable, `p`, which is assigned the value of the pointer returned by the above function, `scopy()`.

```
/* File: allocerr.c - continued */
/* PROGRAM BUG */
main()
{    STRING p,  scopy(STRING t);

     p = scopy("hello");
     puts(p);
}
```

The function, `scopy()`, returns a pointer to an array which has already been freed for other uses. The now freed memory, previously holding the array, must be assumed to have garbage value. The pointer to this garbage is assigned to `p`. The function, `puts()`, assumes `p` is a valid string and will print whatever garbage `p` points to, not the original meaningful string. Without a clear understanding, the above type of error is hard to pinpoint. The freed memory holding the array may or may not be immediately used for other purposes; thus, sometimes, `puts()` in the above example may print a (partly) meaningful string. At other times, it will print out all garbage.

The only solution is to declare all the needed arrays in the calling function, `main()` and pass them as arguments to called functions. The called functions can then put strings in these arrays and the calling function, `main()`, can later use these strings without any problem. The correct structure is as follows:

```
...
void scopy (STRING s, STRING t);
```

```
main()
{   char s[SIZE], t[SIZE];

    scopy(s, t);
    ...
}
```

Using String Functions with Substrings

The function, `strcpy()`, is given two character pointers, one to the destination array and one to the source string. These pointers may point to any character position within an array which corresponds to a *substring* beginning at that position, continuing to the next `NULL` in the array. We can call our string functions with arguments that are substrings of other strings. For example, we can copy a substring of `t` into any location in `s`:

```
/*  File: partstr.c
    Program shows overwriting part of a string with part of another.
*/
#include <stdio.h>
#include <string.h>
#define SIZE 100

main()
{   char s[SIZE], t[SIZE];

    printf("***Partial Strings***\n");
    strcpy(s, "This can be trouble");
    strcpy(t, "Insert string");

    printf("Old s: "); puts(s);
    printf("Old t: "); puts(t);

    strcpy(s + 3, t + 5);
    printf("New s: "); puts(s);
}
```

Sample Session:

```
    ***Partial Strings***
    Old s:  This can be trouble
    Old t:  Insert string
    New s:  Thit string
```

The program copies a substring starting at `t + 5` into a location pointed to by `s + 3`. String copy terminates with a `NULL`; any remaining characters in string `s` after the first `NULL` are not part of the string.

We can even use `strcpy()` to copy part of a string to a different location in the string itself. As always, we must be sure that we are dealing with `NULL` terminated strings and must also take care

that the copy process does not overwrite useful data. For example, here is a loop that eliminates leading white space from a string, `s`:

```
strcpy(s, "    Aloha");
while (isspace(*s))
    strcpy(s, s + 1);
```

The function, `isspace()`, is a library routine that returns True if the argument is a white space. (We have indicated white space explicitly by a *). The loop is executed as long as `*s`, the first character of `s`, is a space. In the loop, the string starting at `s + 1` is copied into `s`, character by character. Each time the loop is executed, one leading white space is removed from `s`. Here are the successive strings starting with the original (again we use white space indicator *).

```
****Aloha
***Aloha
**Aloha
*Aloha
Aloha
```

When a string is copied into itself by `strcpy()`, as long as destination index is less than the source index, we overwrite only the desired characters. If the destination index is greater than the source index, destination characters will be overwritten. For example:

```
strcpy(s, "abcdef");
strcpy(s+1, s);
```

The second `strcpy()` copies `s[0]`, i.e. 'a' into `s[1]`; then copies `s[1]` into `s[2]`; then copies `s[2]` into `s[3]`; etc. All elements of `s` are overwritten with 'a', even the `NULL`, resulting in a non-valid string — a logic error.

Next, let us consider moving the `NULL` position. Since the first `NULL` terminates a string, we can move the `NULL` to squeeze out unneeded trailing characters. Here is a loop that eliminates trailing white space:

```
    while (isspace(s[strlen(s) - 1]))
        s[strlen(s) - 1] = NULL;
```

Starting with the original, successive strings are shown below with an explicit terminating `NULL` (again, we use a * as a white space indicator):

```
Aloha****\0
Aloha***\0
Aloha**\0
Aloha*\0
Aloha\0
```

11.2.3   String Operations: *strcmp()* and *strcat()*

In the last section we saw how a string can be copied and how to determine the length of a string. Two other common operations on strings are to compare them and to join strings, i.e. *concatenate* them.

Our next task is to read lines of text, until a blank line is entered, and examine each line to see if it is the same as a "control string". If a line equals the control string, the line is ignored; otherwise, it is appended to a buffer. When a blank line is encountered in the input, the buffer is printed. The control string is assumed to be entered as the first line. Here is the task:

JOIN: Read a first line as the control string. Read other lines until a blank line is entered, either adding each line to a buffer or discarding it. A line is discarded if it equals the control string. When a line is added to the buffer, separate it from the previous text by a space. Print the buffer at the end of input.

The algorithm will require several functions: one to compare strings, another to append (i.e. concatenate) one string to another. Here is the algorithm:

```
    initialize the buffer to an empty string
    read the first line into the control string

    while not a blank line, read a line
        if the new line is not equal to the control line
            then if the buffer is not empty, append a space to the buffer
                append the new line to the buffer

    print the buffer
```

The two new string operations we will need are provided by the standard library. We will use them to implement our algorithm. The first function compares two strings:

```
    int strcmp(STRING s1, STRING s2);
```

The function, `strcmp()`, compares the strings, `s1` and `s2`, and returns an integer indicating the result of the comparison. If the two strings are equal, it returns a zero value. If the two strings are not equal, the function returns the difference between the first two unequal characters in the two strings. The returned value will be positive if `s1` is *lexicographically* greater than `s2`, and negative if `s1` is less than `s2`. Thus, the `strcmp()` function is the equivalent of a *relational* operator for strings.

The second function we need is to join two strings. Again, the standard library provides a function:

```
    STRING strcat(STRING s1, STRING s2);
```

which *concatenates* (i.e. joins) the two strings, `s1` and `s2`, and stores the result in `s1`. It returns `s1`, i.e. the pointer to the combined string. This is the equivalent of the *addition* operator for strings. The prototypes for these and other standard library string functions are in a header file, `string.h`.

We can now use these functions to implement our program as shown in Figure 11.4. We first

```
/*  File: text.c
    Program reads strings until a blank line is entered. The first string
    read is used as a control. If the other strings are not equal to the
    control string, they are concatenated to the buffer but separated by a
    space. It prints out the buffer at the end. A debug statement prints the
    accumulated string at each step and its length.
*/

#include <stdio.h>
#include <string.h>
#define SIZE 100
#define DEBUG

main()
{   char s[SIZE], control[SIZE], text[SIZE];

    printf("***Build Text: Exclude Control String***\n\n");
    printf("Type control string: ");
    gets(control);
    strcpy(text, "");   /* length of empty string is zero */
    printf("Type text lines, RETURN to quit\n");

    while (*gets(s)) {
        if (strcmp(s, control) != 0) {

            if (strlen(text))
                strcat(text, " ");

            strcat(text, s);
        }

        #ifdef DEBUG
            printf("debug:length of buffer is %d: %s\n",
                    strlen(text), text);
        #endif
    }

    printf("Final buffer is: ");
    puts(text);
}
```

Figure 11.4: Code using strcmp() and strcat()

read a string into the variable, `control`, and initialize the buffer, `text`, to an empty string. The while loop then reads strings until a blank line is entered. Since the expression `gets(s)` reads a line of text and returns the destination pointer, `s`, `*gets(s)` is the first character of the string read into `s`. The expression is True if any non-empty string is entered. It is False when the first character of `s` is a `NULL` which occurs when an empty line (just a `RETURN`) is entered.

For each string read into `s`, we compare it with `control`. If they are not equal, we concatenate `text` and `s`. A space is concatenated to `text` if it is not empty, so that the concatenated strings are separated by a space. We have included a debug statement to print the accumulated buffer and its length. When the input terminates, the accumulated string, `text`, is printed. Here is a sample session:

```
***Build Text:  Exclude Control String***

Type control string:   Hello
Type text lines, RETURN to quit
Hello
debug:length of buffer is 0:
earth
debug:length of buffer is 5:   earth
calling
debug:length of buffer is 13:   earth calling
moonbase,
debug:length of buffer is 23:   earth calling moonbase,
hello
debug:length of buffer is 29:   earth calling moonbase, hello

Final buffer is:   earth calling moonbase, hello
```

Observe that string comparisons are *case distinct*, e.g. *hello* is not the same as *Hello*, so the first *Hello* in the input is discarded, while the second, *hello*, is not.

The function, `strcmp()`, can be used when we wish to search for a particular string or when we wish to order strings in lexicographic or dictionary order. Unfortunately, upper case and lower case values of a letter are not equal as shown above; therefore, we must change all strings to the same case (e.g. by using `tolower()`) for a case independent comparison.

To understand how these library functions work, let us write our own versions of functions `strcmp()` and `strcat()`, beginning with `our_strcmp()`. First, let us look in a little more detail of "what" `strcmp()` does. Given two strings, the comparison proceeds character by character until two unequal characters are encountered, or both the strings are exhausted. When two unequal characters are encountered, their difference is returned. If no unequal characters have been encountered when both strings have reached `NULL`, the two strings are identical, and zero is returned. Here are some examples of results using `strcmp(string1, string2)`:

```
/*  File: str.c - continued
    Compares strings s and t, returns difference of first
    unequal characters or returns zero.
*/


int our_strcmp(STRING s, STRING t)
{
    while ( *s ) {                  /* terminate when s  is exhausted    */

        if (*s != *t)           /* if unequal, break loop            */
            break;

        s++;                    /* traverse the two strings          */
        t++;
    }

    return *s - *t;             /* return the difference of characters */
}
```

Figure 11.5: Code for our_strcmp

string1	string2	returned value	comment
hawaii	hawaiian	0 - 'a'	negative
hilo	hawaii	'i' - 'a'	positive
hawaii	hawaii	0	zero
hawhaw	hawaii	'h' - 'a'	positive
Hawaii	hawaii	'H' - 'h'	negative
haw123	hawaii	'1' - 'a'	negative

We can model our algorithm on this behavior of strcmp(). We traverse both strings until we arrive at a terminating NULL in either one. During traversal, we examine the corresponding characters in the strings to see if they are unequal. If so, we terminate the traversal loop. Otherwise, we continue the process. When the loop is terminated, we return the difference between the characters where we left off in the two strings.

Figure 11.5 shows the code implementing this algorithm. The while loop traverses strings s and t terminating when s points to a NULL character. Within the loop, the corresponding characters of the two strings are compared. If unequal characters are encountered, the loop is terminated, and the difference between the characters is returned. If the loop terminates because *s is zero, then no unequal characters have been encountered so far, but the string t may or may not be exhausted. In either case, *s - *t, i.e 0 - *t is returned. In particular, if t points to NULL (the string t is also exhausted), then the two strings are equal and zero is returned. Otherwise, the difference between the first unequal characters is returned. Note, we do not need to test for the end of the string t in the while condition. If t terminates before s, then the NULL at the end of string t will not compare equal to *s, and the loop will terminate anyway.

```
/*  File: str.c - continued
    Concatenates s and t by appending t to s. Returns
    pointer to s. s must point to a large enough array to accommodate the
    concatenated string.
*/

STRING our_strcat(STRING s, STRING t)
{   STRING p;

    p = s;                  /* save pointer s */

    while (*s)              /* increment s until it points to NULL */
        s++;

    strcpy(s, t);           /* copy t into s */
    return p;               /* return saved pointer */
}
```

Figure 11.6: Code for `our_strcat()`

To write `our_strcat()`, we must append the second string to the end of the first string; so we must traverse the first string until we find the `NULL`. We can then copy the second string at this point in the first using `strcpy()`. The function returns the pointer to the destination string, i.e. the beginning of the first string. Since the function must return a pointer to the original string, `s`, we save the original pointer in a variable, `p`. We then increment `s` until it points to the terminating `NULL`. We then copy `t` into `s` starting at the `NULL` character position using `strcpy()`, and return the saved pointer, `p`. This function performs the same task as does `strcat()`.

11.2.4   String Conversion Functions

Besides the functions for manipulating strings discussed in the previous sections (and others not discussed, but presented in Appendix C), the standard library provides several functions for converting the character (ascii) information in a string to other data types such as integers or floats.

We will illustrate the use of one such function, `atoi()`, by modifying our function `getint()` that we wrote in Chapter 4. Recall, this function reads the next valid integer from the standard input character by character, skipping over any leading white space, converts the character sequence to an integer representation, and returns the integer value. The prototype for this function is:

```
int getint(void);
```

In our previous version of this function, we made it robust enough to detect when `EOF` or invalid (non-digit) characters are present in the input. Here we will extend the utility of `getint()` to read the next white space delimited item in the input, and convert it to integer form, this time allowing a leading + or - sign, and give the user the opportunity to re-enter data for illegal character errors.

GETINT: Write a program that reads only a valid integer. If there is an error in entering an integer, it detects the error and allows the user to re-enter the data.

The program driver is quite simple; it calls the function `getint()` that returns a valid integer read from the standard input. The driver then prints the integer returned by the function. Here is the algorithm for getint():

```
initialize valid to False

while not a valid string
    read a string s
    set valid to True if s represents a valid integer

    if valid
        return an integer represented by the string s
    else
        print an error message
```

The function reads in the input as a string, and checks if it is a valid digit string for an integer. To check if a string `s` is a valid integer string, we examine whether it consists of only digits with, perhaps, a leading unary sign (`+` or `-`). The following algorithm sets `valid` to True if `s` represents a valid integer:

```
if *s is '+' or '-'
    valid = digitstr(s + 1);
else
    valid = digitstr(s);
```

If the first character of `s` is a unary sign, check the rest of the string (starting as `s + 1`) for all digits; otherwise check the entire string `s` for all digits.

If `s` is a valid digit string, the function returns an equivalent integer using the standard library function, `atoi()`. The call `atoi(s)` returns the integer represented by the string `s`. The function atoi() has the prototype (included in `stdlib.h`):

```
int atoi(STRING s);
```

If `s` is not a valid string, the user is prompted to type the input again.

To check if all characters in a string are digits, `getint()` uses the function `digitstr()`. The algorithm modules are combined and implemented in a program shown in Figure 11.7.

The driver gets an integer and prints it. The function `getint()` reads the next white space delimited string in the input using `scanf()`. If the first character is a unary operator, we check for digits string starting at the pointer `s + 1`; otherwise, we check starting at the pointer `s`. The flag, `valid`, stores the value returned by `digitstr()`. If `valid` is True, we use `atoi()` to return the integer represented by `s`; otherwise, we print a message prompting the user to re-enter the integer, flush any remaining characters on the input line, and read the new input. The flag `valid` is initialized to False, and the loop continues as long as `valid` remains False, i.e. as long as a valid integer is not entered.

The function `digitstr()` traverses the string until a `NULL` appears. If a non-digit is encountered anytime during traversal, it returns False; otherwise, at the end of traversal, it returns True. It uses the library function, `isdigit()` to check if a character is a digit.

A sample session is shown below:

```
/*  File: intchk.c
    This program reads and prints an integer. It detects errors in
    input and asks the user to retype.
*/
#include <stdio.h>
#include "tfdef.h"        /* defines TRUE, FALSE *.
#include <stdlib.h>       /* prototype for atoi() */
#include <ctype.h>
#define SIZE 100
main()
{   int n;
    printf("***Valid Integer Input***\n\n");
    printf("Type an integer: ");
    n = getint();
    printf("Integer is %d\n", n);
}
/* Function gets a valid integer. */
int getint(void)
{   char s[SIZE];
    int valid = FALSE;         /* flag for valid string */

    while(!valid) {
        scanf("%s",s);                    /* read a string delim by ws */
        if (*s == '+' || *s == '-')   /* if first char is + or -, */
            valid = digitstr(s + 1); /* check rest of string; */
        else                              /* otherwise, */
            valid = digitstr(s);      /* check the entire string */

        if (valid)                   /* if a valid string */
            return(atoi(s));     /* return its equivalent integer */
                                      /* otherwise, */
        printf("***Error in input\n");/* print an error mesg */
        printf("Re-enter your integer: ");
        while(getchar() != '\n'); /* flush remainder of input */
    }
}
/* File: str.c - continued */
/* Checks if a string t is all digits  */
int digitstr(STRING  t)
{
    while (*t)
        if (!isdigit(*t))        /* if any character in t is */
            return FALSE;        /* NOT a digit, return FALSE */
        else  t++;               /* else point to next char. */
    return TRUE;         /* if all chars are digits, return TRUE */
}
```

Figure 11.7: Code for getint()

```
***Valid Integer Input***

Type an integer:   123e
***Error in input
Re-enter your integer:   =123
***Error in input
Re-enter your integer:   -+123
***Error in input
Re-enter your integer:   -123
Integer is -123
```

11.2.5   File I/O with Strings

Earlier in this chapter, we described library functions to do string I/O with the standard input and output. The library also provides functions to do I/O with files. Here we will illustrate the use of these functions with our next task; to search for the presence of a string in the lines of a text file.

GETLNS: Search for a control string in the lines of a file. Each line that contains the control string is to be written to an output file and to the standard output.

The algorithm is written easily if we write a function, **srchstr()**, that searches for the presence of one string in another. Here is the algorithm:

```
get the control string control
open files

while not EOF, read a line s from the input file
    if srchstr(s, control) is True
    then write the line to output file and stdout
```

We could use character I/O to read from an input file, but it is easier to use library string I/O functions: **fgets()** and **fputs()**.

```
int fgets(STRING s, int n, FILE *fp);
int fputs(STRING s, FILE *fp);
```

These functions are similar to **gets()** and **puts()** with minor differences. The function **fgets()** reads a string from a stream, **fp**, into a buffer, **s**. The maximum size, **n**, of the string buffer must be specified to **fgets()** and must allow for a terminating **NULL** character. The function reads a string until a newline character is encountered or the specified maximum size of buffer is reached. It adds the terminating **NULL**, but it does **NOT** strip the newline character as does **gets()**. The **NULL** is added **after** the \n character and **fgets()** returns the buffer pointer if successful, or **NULL** otherwise.

The function **fputs()** outputs a string to a stream **fp**. It strips the terminating **NULL** from the string, but does **NOT** add a newline character as does **puts()**. The function returns the last character output if successful, **EOF** otherwise. The prototypes for these functions are included in **stdio.h**.

```
/*  File: srchstr.c

    This program searches for a string in an input file. Every line
    that contains the string is printed out.
*/
#include <stdio.h>
#include "tfdef.h"
#include "strtype.h"
#define SIZE 100

main()
{   FILE *input, *output;
    char infile[15], outfile[15];
    char s[SIZE], control[SIZE];

    printf("***String Search***\n\n");
    printf("Type a string to be searched for: ");
    gets(control);

    printf("Input file : ");
    gets(infile);
    printf("Output file : ");
    gets(outfile);
    input = fopen(infile, "r");
    if (input == NULL) {
        puts("*** Can't open input file ***");
        exit(0);
    }
    output = fopen(outfile, "w");
    if (output == NULL) {
        puts("*** Can't open output file ***");
        exit(0);
    }

    while (fgets(s, SIZE - 1, input))

        if (srchstr(s, control)) {
            puts(s);
            fputs(s, output);
        }

    fclose(input);
    fclose(output);
}
```

Figure 11.8: Driver for Text Searching Program

```
/*  File: srchstr.c - continued */
/* Function tests if str is in s */
int srchstr(STRING s, STRING str)
{

    while (*s)
        if (compare(s, str))        /* if str is at the start of s */
            return TRUE;            /* return True */
        else s++;                   /* otherwise, go to the next pos. */

    return FALSE;                   /* string exhausted, return False */
}
```

Figure 11.9: Code for `srchstr()`

The program driver for our task is easy to write as shown in Figure 11.8. The program driver first reads the control string to search for. It then opens the input and output files. The while loop reads lines from the input file until end of file. Each line read is tested by `srchstr()` for the presence of the control string. If the control string is present in the line, it is written to both `stdout` and the output file. We will need `TRUE` and `FALSE` definitions for `srchstr()`, so we have included the header file, `tfdef.h`.

The function, `srchstr()` traverses the string, `s`, and tests if the control string is present at each position in `s`. If it is present, it returns `TRUE`; otherwise, it goes to the next position. The function, `srchstr()`, uses a function, `compare()`, to see if a string is present at the start of another string. This is different than `strcmp()` since the string we are searching in may not terminate at the end of the control string. The code for `srchstr()` is shown in Figure 11.9. Given a string, `s`, and a control string, `str`, it starts at the first character of `s`, and calls `compare()` to see if `str` is present in `s` starting at the first character. If it is, it returns `TRUE`; otherwise, it increments `s` to point to the next character. If the string is exhausted, it returns `FALSE`.

The code for `compare()` is shown in Figure 11.10. It traverses `str` and `s` until `str` is exhausted. If it encounters corresponding characters that are not the same in the two strings, it returns `FALSE`. When `str` is exhausted, it returns `TRUE`. Here is a sample session:

```
***String Search***

Type a string to be searched for:  while
Input file :  ucstr.c
Output file :  xyz
while (gets(s)) {
```

The file `ucstr.c` contains only one line with the string *while* in it. That line is written to the file `xyz` and to `stdout`.

For this task, we have written our own function to compare `str` with the first several characters in string `s` because we do not expect `s` to terminate at the end of the control string, `str`. If `n` is

```
/*  File: srchstr.c - continued */
/* Function tests if str is at the start of s */

compare(STRING s, STRING str)
{
    while (*str)
        if (*str++ != *s++)
            return FALSE;

    return TRUE;
}
```

Figure 11.10: Code for `compare()`

the length of str, then we require a comparison of the first n characters in the two strings. There is a standard library function, strncmp(), which does just that:

```
    int strncmp(STRING s, STRING t, unsigned n);
```

It compares the first n characters of s and t, and returns the difference of the first unequal characters, or it returns zero if they are all equal, just like strcmp(). So, instead of compare(s, str), we could have used:

```
    strncmp(s, str, strlen(str))
```

A similar library function, strncpy(), is also available:

```
    STRING  strncpy(STRING s, STRING t, unsigned n);
```

which copies n characters from the source string, t, into the destination string, s, without adding a terminating NULL. It returns s.

We close this section by emphasizing the difference between gets(s) and fgets(s, n, fp). Let us assume an input string "Hawaii\n" is in the standard input, and that the string s is large enough to accommodate the example string with n selected appropriately. The string, s is shown below after the use of each function:

```
    gets(s):            Hawaii\0     /* newline stripped, NULL appended */
    fgets(s, n, stdin):  Hawaii\n\0   /* newline present, NULL appended */
```

Similarly, the output of the functions puts(s) and fputs(s, fp) is shown below:

```
    puts(s):            Hawaii\n  /* NULL stripped, newline appended */
    fputs(s, stdout):    Hawaii\n  /* NULL stripped */
```

11.3 More Example Programs

In the previous section we have discussed some of the string utility functions provided by the C standard library and illustrated their use with examples. Additional string functions can be found in Appendix C. We close this chapter with a few additional example programs making use of these string processing functions.

11.3.1 Palindromes

Our next task is:

PALI: Read strings and check if each is a palindrome.

A palindrome is a string that reads the same forwards and backwards, for example:

<div align="center">able was i ere i saw elba</div>

The algorithm is simple: compare the reverse of the string with the original string. If they are the same, the string is a palindrome.

```
while not EOF, read a string s
    copy reverse of s into t

    if s and t are equal,
        s is a palindrome
    else
        s is not a palindrome
```

The driver follows the algorithm closely, as seen in Figure 11.11. We will use a function, revcpy(), to copy the reverse of the string.

We must write the function revcpy() to copy one string into another in reverse order. To see how the algorithm for this function should proceed, we will work with the indices in the source and destination strings as shown below:

<div align="center">

src: hello\0
sind: <—4

dest: olleh\0
dind: 0—>

</div>

The string, src, is shown with the terminating NULL and the source index, sind, must start at the last character of src and decrease as each character is copied. In the destination string, dest, the index, dind, must start at 0 and increase as each character is copied. When the source index becomes negative, all characters have been copied in reverse order from the source. After all the characters are copied, a terminating NULL must be added to the destination string. Here is the algorithm:

```
initialize sind to the last index of src and dind to 0
while sind is >= 0
    copy from src to dest
    increment dind and decrement sind
add a NULL to dest
```

```
/*   File: pali.c
     Program reads a string and tests whether it is a palindrome.
     It repeats the process until EOF.
*/

#include <stdio.h>
#include <string.h>
#include "strtype.h"
#define SIZE 100

main()
{   char s[SIZE], t[SIZE];

    printf("***Palindrome Test***\n\n");
    printf("Type strings, EOF to quit\n");

    while (gets(s)) {
        revcpy(t, s);                    /* copy reverse of s into t */

        if (strcmp(s, t) == 0)
            printf("%s: a palindrome\n", s);
        else
            printf("%s: not a palindrome\n", s);
    }
}
```

Figure 11.11: Driver for Palindrome

```
/*  File: pali.c - continued
    Function copies string src in reverse order into string dest.
*/

void revcpy(STRING dest, STRING src)
{   int sind, dind = 0;        /* dest index is 0 */

    sind = strlen(src) - 1;  /* source index at last character */

    while (sind >= 0)          /* loop while source index is non-neg. */
        dest[dind++] = src[sind--];   /* copy character, and update */

    dest[dind] = NULL;                 /* append a NULL */
}
```

Figure 11.12: Code for `revcpy()`

The function is shown in Figure 11.12.

Here is a sample session:

```
***Palindrome Test***

Type strings, EOF to quit
```
this is it
```
this is it:  not a palindrome
```
able was i ere i saw elba
```
able was i ere i saw elba:  a palindrome
```
^D

Our function, `revcpy()`, will work fine as long as the source and destination strings are different strings. We could write a function to reverse a string in place. We can follow the same procedure of copying from the source index to the destination index; however, since the source and destination strings are the same string, characters at source index as well as at destination index must be swapped rather than simply assigned. Otherwise, copying a character from the source index to the destination index will overwrite a character.

$$
\begin{array}{rl}
\text{s:} & \text{hello}\backslash 0 \\
\text{sind:} & <\!\!-\!\!4
\end{array}
$$

$$
\begin{array}{rl}
\text{dind:} & 0\!-\!\!> \\
\text{new s:} & \text{olleh}\backslash 0
\end{array}
$$

Furthermore, the characters need only be swapped as long as source index is greater than destination index. When the source index is less than the destination index, all characters have been swapped. If the two indices are equal, the corresponding characters are the same and need no swapping. Finally, a terminating NULL need not be added since it is already present in the correct position. Figure 11.13 shows the code for the function `revself()`.

```
    /*  File: str.c - continued
        Function reverses string s in place.
    */
    void revself(STRING s)
    {   int c, sind, dind = 0;   /* c used as temp. storage during a swap */
                                 /* dest index at 0 */

        sind = strlen(s) - 1;    /* src index at last char. */

        while (dind < sind) {    /* loop while chars need swapping */
            c = s[dind];         /* swap characters, */
            s[dind++] = s[sind]; /* and update indices */
            s[sind--] = c;
        }
    }
```

Figure 11.13: Code for `revself()`

11.3.2   Words

Our next task is to break up a string into words delimited by white space.

WDS: Read strings; break up each string into its constituent words.

The algorithm starts by skipping over leading white space. If the string is not exhausted, a word starts at that position and continues until the next white space. Here is the algorithm.

```
while not EOF, read a string s

   initialize string index to 0

   while not NULL
        skip over leading white space
        initialize word index to 0

        copy the next word into wd
        terminate word with a NULL
        print the word
```

In our algorithm, a word is any sequence of characters delimited by white space. Figure 11.14 shows the program. It reads lines until EOF scanning each line until a NULL is reached. Each scan first skips over white space, then copies a word into a string, wd, while characters are non-white space and non-NULL. A terminating NULL is added to the word and it is printed. Here is sample session:

```
    ***Words in Strings***
```

```
/*  File: strwds.c
    This program reads strings until EOF. For each string read, it copies each
    of the words into another string and prints it.
*/

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define SIZE 80
#define WDSIZE 40

main()
{   char s[SIZE], wd[WDSIZE];
    int i, j;

    printf("***Words in Strings***\n\n");
    printf("Type strings, EOF to quit\n");

    while (gets(s)) {                     /* read lines until EOF */
        i = 0;

        while (s[i]) {                    /* repeat while s[i] is not NULL */

            while (isspace(s[i]))    /* skip leading white space */
                i++;

            j = 0;                   /* initialize for a new word */

            while (s[i] && !isspace(s[i])) /* while non-NULL AND non-white */
                wd[j++] = s[i++];          /* copy word */

            wd[j] = NULL;                 /* terminate string */
            puts(wd);                     /* print word */
        }
    }
}
```

Figure 11.14: Code for separating words from a string

```
Type strings, EOF to quit
```
This is a test
```
This
is
a
test
```
^D

11.3.3   Substrings

In string manipulations, it is frequently necessary to find a substring of a string. A substring is a string that is part of another string. It can be parameterized by specifying where the substring starts and how long it is. Our next task is to write a program that finds a substring of a string at a given position and of a specified size.

SUBSTR: Read substring parameters. For each line of input, find the appropriate substring. For example, consider the string:

Source string: This is a test string\0

(The terminating NULL is shown explicitly). A substring of this string starting at index position 2 and containing 5 characters is:

Destination string: is is\0

We will write a function to extract such a substring. The function must be passed several arguments: the source string (pointer), a destination string where the specified substring is to be copied (pointer), the starting position of the substring in the source string (integer), and the number of characters for the substring (integer).

It may or may not be possible for the function to extract the string. For example, if the starting position is outside the string, no substring can be extracted. We will assume that the function returns the destination string (pointer) if successful in extracting a string; otherwise, it returns a NULL pointer to indicate failure. We will also assume that the function will extract as many characters as possible upto the specified number. The function prototype should be:

```
STRING substr(STRING src, STRING dest, int startpos, int nchrs);
```

The parameter, src, points to the source string, and dest points to an array where the substring of src is to be copied. The next two arguments provide the starting position and the number of characters. The **calling** function must allocate memory for the destination string. The starting position, startpos is an index into the array — it must be 0 or greater. The parameter, nchrs is the maximum number of characters to copy into the substring.

Since the program depends primarily on substr(), let us first develop an algorithm for it. The function must start copying characters from the starting position startpos. If we use an array index, src[startpos] accesses the character at the start position if startpos is in the source string. If startpos is not in the source string, we will return a NULL to indicate failure to extract a substring.

Next, we must copy up to a maximum of nchrs characters into dest. When the source string is exhausted or nchrs characters are copied, we stop the copy process and append a NULL to the

substring. If even one character is copied into the substring, we will return the destination pointer. Here is the algorithm:

```
if startpos >= strlen(src)
    return NULL

j = 0;
while j is less than nchrs and src is not exhausted
    copy a character: dest[j] = src[startpos + j]
    increment j

terminate dest with NULL
return dest
```

The program driver reads the start position and the number of characters. It then reads strings until end of file and finds the substring for each string if possible. The code for the driver and `substr()` is shown in Figure 11.15. The program prints the substring if it can be extracted; otherwise, it prints a message. Here is a sample session:

```
***Substring Extraction***

Type start position and number of characters:   2 5
Type text lines, EOF to quit
this is a test string
is is
hello
llo
well
ll
he
Substring cannot be extracted
then
en
^D
```

11.4   Common Errors

1. Failure to include library header files, e.g. `string.h`. Prototypes for library string routines are not included resulting in default assumptions and consequent problems.

2. We have already discussed common string related errors in Chapter 7 and in this chapter. Always allocate space for an array where a string is to be stored. Once space is allocated for a string, pointer variables can be used to access strings.

3. Array names must not be used as Lvalues.

```
/*  File: substr.c
    Program extracts a substring and prints it.
*/
#include <stdio.h>
#include "strtype.h"
#define SIZE 80
STRING substr(char src[], char dest[], int startpos, int nchrs);

main()
{   char s[SIZE], sub[SIZE];
    int start, n;

    printf("***Substring Extraction***\n\n");
    printf("Type start position and number of characters: ");
    scanf("%d %d%*c", &start, &n);       /* suppresses newline */
    printf("Type text lines, EOF to quit\n");

    while (gets(s)) {
        if (substr(s, sub, start, n)) /* if substring, */
            puts(sub);                    /* print it */
        else
            printf("Substring cannot be extracted\n");
    }
}


/*  Function copies a substring of src, starting at i and n characters
    long, into dest. It returns dest if success; NULL otherwise.
*/
STRING substr(STRING src, STRING dest, int startpos, int nchrs)
{   int j;

    if (startpos >= strlen(src))
        return NULL;

    for (j = 0; j < nchrs && src[startpos + j]; j++) {
        dest[j] = src[startpos + j];
    }

    dest[j] = NULL;
    return dest;
}
```

Figure 11.15: Code for the substring program

11.5 Summary

This chapter has discussed a very common data type in C programs: the string. We have briefly introduced the concept of an *abstract data type* as consisting of a data declaration and a set of operations on data items of that type. We have defined a user defined type, STRING, for string data and used it throughout the chapter. (While our string data type does not completely satisfy the definition of an abstract data type, the basic concept is seen).

Many common operations on string data are provided through the standard library. We have described a few of these; in particular functions for I/O: gets() and puts(), and file I/O: fgets() and fputs() whose prototypes are defined in stdio.h. In addition the functions for string manipulation, strlen() and strcpy() as well as string operation, strcmp() and strcat(), have been described. Other functions described include atoi(), strncmp(), and strncpy().

Throughout the chapter we have shown numerous examples of programs for string processing.

11.6 Exercises

1. If the characters in an array, `s` are: string\0 of characters\0

   What does each of the following print? Show each character.

   ```
   printf("%s", s);
   puts(s);
   fputs(s, stdout);
   ```

2. If the input of characters is:

   ```
   string of characters\n
   ```

   What does each of the following read? Show each character, including `NULL`.

   ```
   scanf("%s", s);
   gets(s);
   fgets(s, sizeof(s) - 1, stdin);
   ```

3. Assume `s` is a string array. Under what condition is each of the following True?

   ```
   s
   *s
   !*s
   gets(s)
   *gets(s)
   !*gets(s)
   ```

   Find and correct any errors in the following and determine the outputs where feasible. The input is shown when appropriate.

4. ```
   main()
   {   char s[80], t[80];

       s = "this is a message";
       if (s == t)
            printf("Equal\n");
       else
            printf("Not equal\n");
       puts(s);
   }
   ```

5. ```
main()
{   char s[80], t[80];

    scanf("%s", s);
    printf("%s", s);
}
```
Input: This is a message

6. ```
main()
{   char *s;

    s = "this is a message";
    printf("%d  %s\n", s, s);
    puts(s);
}
```

7. ```
main()
{   char *s;

    gets(s);
    puts(s);
}
```

8. ```
main()
{   char s[80];

    while (*s) {
        putchar(*s);
        s++;
    }
}
```

9. ```
main()
{   char *s;

    strcpy(s, "hello");
    puts(s);
}
```

10. ```
main()
{   char name[80];

    name = get_str(name);
    puts(s);
}

char *get_str(char *s)
```

```
    {
        gets(s);
        return s;
    }

11. int cmpstr(char *s, char *t)
    {
        if (s == t)
            return TRUE;
        else
            return FALSE;
    }
```

11.7 Problems

Write program drivers for each of the following. The driver should read appropriate data until end of file, call the functions described below, and print the results.

1. Write a function that returns the index where a character, `c`, occurs in a string, `s`. The function returns -1 if `c` is not present in `s`. Use array indexing.

2. Repeat 1 using pointers.

3. Write a function that returns the index where a character, `c`, occurs in a string `s`; the search for `c` starting at a specified index, `i` in `s`. The function returns -1 if `c` is not present in `s` starting at the index, `i`. Use array indexing.

4. Repeat 3 using pointers.

5. Write a function, `how_many()`, that returns the number of times a character, `ch`, occurs in a string, `s`. Use array indexing.

6. Repeat 5 using pointers.

7. Write a function that substitutes a new character, `newc`, for the first occurrence of a character, `c`, in a string, `s`. Use array indexing.

8. Repeat 7 using pointers.

9. Write a function that substitutes a new character, `newc`, for every occurrence of a character, `c`, in a string, `s`. Use array indexing.

10. Repeat 9 using pointers.

11. Rewrite the function, `our_strcpy()` in Section 11.2.2 so that it properly returns the pointer to the destination string.

12. Write a function that takes two strings, `s` and `t`, as arguments. Copy string `s` into `t`, but remove all white space and punctuation. Use array indexing.

13. Repeat 12, but use pointers.

14. Write a function that takes a string of characters and removes all white space and punctuation in that same string. Use array indexing.

15. Repeat 14 using pointers.

16. Write a function, `xwslead()`, that removes all leading white space from a string. Use array indexing.

17. Repeat 16 using pointers.

18. Write a function, `xwstrail()`, that removes all trailing white space from a string. Use array indexing.

19. Repeat 18 using pointers.

20. Write a function, `xws()`, that removes all leading and trailing white space from a string. Use array indexing.

21. Repeat 20 using pointers.

22. Write a function, `squeeze()`, that removes all white space from a string. Use array indexing.

23. Repeat 22 using pointers.

24. Write a function, `compare()`, that takes two strings as arguments and compares them for equality after leading and trailing blanks are removed. If the strings are equal after the leading and trailing blanks are removed, the function returns True. Otherwise, it returns False.

25. Write a function, c̄ompstrip(), that takes two strings as arguments and compares stripped versions of them. A stripped string is one from which all white space and punctuation are removed. Function returns True if the strings are equal after they are stripped.

26. Write a function, `palindrome()` that checks if a given string is the same forwards and backwards. Use pointers.

27. Write a function that checks if a string is a palindrome ignoring all white space. Example:

    ```
    i ia wah hawaii
    ```

28. Write a function that takes two string arguments, `s` and `t`. Copy `s` into `t` in reverse order, except that a sequence of white space is squeezed to a single space.

29. Write a function that takes a single string argument, and reverses the string itself, except that white space is squeezed to a single space.

30. Write a function that removes the first word from a string. Write a program that uses the function to remove a specified number of leading words from a string.

31. Write a function that removes the last word in a string. Write a program that uses the function to remove a specified number of trailing words from a string.

32. Write a function that takes two strings, `s1` and `s2` as arguments. It returns the index where `s2` occurs in `s1`, or it returns -1 if `s2` is not in `s1`.

33. Write a function that substitutes a new string, `repl_str`, for the first occurrence of a string, `str` in a string, `src`.

34. Write a function that replaces a new string, `repl_str`, for every occurrence of a string, `str`, in `src`.

35. Write a function that detects the presence of a whole word, `wd`, in a string, `s`.

36. Write a function that converts a string into an integer. The conversion is terminated when a non-digit is encountered.

37. Write a function that converts a string into a float. The conversion is terminated when a character that does not belong in a decimal number is encountered.

38. Write a function that converts an integer to a string.

39. Write a function that converts a float to a string.

40. Write a function that converts a string of binary digits to an integer.

41. Write a function that converts an unsigned integer into a string of binary digits.

42. Write a function, `nexttok()`, that gets the next token from a string, starting at a specified array index, called `cursor`. The function returns the new value of `cursor`, the token itself, and the type of the token. Leading white space is skipped. A longest valid token is built as long as the characters belong to a token type. The token is complete when a character that does not belong to a token type being built is encountered.

A valid token type is either an identifier, an integer, a float, an invalid, or an EOS, end of string. An identifier starts with a letter, and may be followed by letters and/or digits. An integer starts with a digit, and may be followed by digits. A float must start with a digit, may be followed by digits, may be followed by a decimal point, and may be followed by a sequence of digits. A character other than white space, letters, and digits is an invalid type token containing that one character. EOS type of token is returned when the `NULL` character is reached.

Write a program that reads in lines of input from a file, and use the above function to print out the tokens in each line until `EOF`.

Chapter 12

Structures and Unions

So far, we have seen one kind of compound (user defined) data type — the array and in Chapters 7 and 9 have seen how we can group information into one common data structure. However, the use of arrays is limited to cases where all of the information to be grouped together is of the same type. In this chapter we present the other compound data type provided in C — the structure, which removes the above limitation. We will discuss structures, pointers to structures, and arrays of structures. As with our previous data types, we will see how such structures can be declared; how information in them can be accessed, and how we can pass and return structures in functions. We will also see how arrays of structures are sorted and searched. We illustrate these points with several example programs.

Finally, we will introduce unions which are similar to structures; however, the elements in the union share the same memory cells. In a union, different types of data may be stored in a variable but at different times.

12.1   Structures

In C, a *structure* is a derived data type consisting of a collection of member elements and their data types. Thus, a variable of a structure type is the name of a group of one or more *members* which may or may not be of the same data type. In programming terminology, a structure data type is referred to as a *record data type* and the `members` are called `fields`. (We will use these two terms interchangebly).

12.1.1   Declaring and Accessing Structure Data

As with any data type, we need to be able to declare variables of that type. In particular for structures, we must specify the names and types of each of the fields of the structure. So, to declare a structure, we need to describe the number and types of fields in the form of a *template*, as well as declare variables of that type. We illustrate with an example: a program that maintains temperatures in both celsius and fahrenheit degrees. A variable, `temp`, is to be used to maintain the equivalent temperatures in both celsius and fahrenheit, and thus requires two fields, both of them integers. We will call one field `ftemp` for fahrenheit temperature and the other `ctemp` for celsius. The program, shown in Figure 12.1, reads a temperature to the `ftemp` field of the variable, `temp`, and uses a function, `f_to_c()`, to convert the temperature from fahrenheit to celsius and store it in the `ctemp` field. In looking at this program, we see that the variable `temp` is declared

```
/*    File: fctemp.c
      Program reads temperature in fahrenheit, converts to celsius, and
      maintains the equivalent values in a variable of structure type.
*/
#include <stdio.h>
main()
{     struct trecd {
            float ftemp;
            float ctemp;
      } temp;
      double f_to_c(double f);
      char c;

      printf("***Temperatures - Degrees F and C***\n\n");
      printf("Enter temperature in degrees F : ");
      scanf("%f",&temp.ftemp);
      temp.ctemp = f_to_c(temp.ftemp);
      printf("Temp in degrees F = %3.1f\n", temp.ftemp);
      printf("Temp in degrees C = %3.1f\n", temp.ctemp);
}

/*    This routine converts degrees F to degrees C */
double f_to_c(double f)
{
      return((f - 32.0) * 5.0 / 9.0);
}
```

Figure 12.1: Code for Simple Structure Program

	ftemp	ctemp
temp		
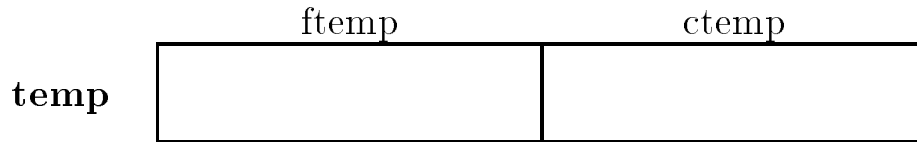
Figure 12.2: Structure Variable `temp` in Memory

to be of structure type with the declaration statement:

```
struct trecd {
     float ftemp;
     float ctemp;
} temp;
```

This statement consists of the keyword, `struct`, followed by the description of the template for the structure and then the variable name. The description of the template, in our example, consists of a *tag* (or name), `trecd` which names the template, followed by a bracketed list of field declarations. The tag is optional. Within its scope, the tag can be used to refer to this structure template without specifying the fields again, explicitly. The bracketed list declares the fields of the structure giving a type followed by an identifier. Our example shows that this structure has two fields: `ftemp` and `ctemp`, both of type `float`.

Figure 12.2 shows the memory cells allocated to the variable `temp`. Two `float` cells have been allocated, one referred to as the `ftemp` field and the other as `ctemp`. The entire block of memory is referred to by the variable name, `temp`. Otherwise, structure declarations are the same as any other variable declaration and have the same scope as would an `int` declaration, for example.

To access the information in a structure, the variable name (in our case, `temp`) is *qualified* using the "dot" operator (`.`) followed by the field name:

```
temp.ftemp
temp.ctemp
```

In general, the syntax for accessing a member of a structure is:

<variable_identifier>.<member_identifier>

In a program, members of a structure variable may be used just like other variables. In the function `main()` above, the argument to `scanf()` is `&temp.ftemp` which is the address of the float cell, `temp.ftemp`. (Precedence of the dot operator is higher than that of the address operator so no parentheses are needed in this case). The numeric value read by `scanf()` will be stored where the argument points — it will be stored in the cell `temp.ftemp`. The rest of the program is straight forward. We have passed a double value to the function `f_to_c` and get a double result which we assign to `temp.ctemp` and print the results.

Sample Session:

```
***Temperatures - Degrees F and C***
```

```
      Enter temperature in degrees F : 78
      Temp in degrees F = 78.0
      Temp is degrees C = 25.6
```

As we have said, the members of a structure variable can be of different types. For example:

```
      struct {
            char name[26];
            int id_number;
      } student;
```

which defines a structure variable, `student`, with two fields: a string of characters called `name`, and an integer called `id_number`. Enough contiguous memory is allocated for the variable student to accommodate both fields. We can find the amount of storage allocated for a structure by using the `sizeof` operator. (Be aware that the total size of a structure variable may not be equal to the sum of the sizes for the fields because of rules about memory alignment which may vary from computer to computer. For example, memory allocation for an integer may have to start at a machine word boundary such as an even byte address. Such alignment requirements may make the size of a structure variable somewhat greater than the sum of the field sizes).

The identifiers used for the field names apply only to variables of that structure type. Different structure types may have fields specified by the same identifier, but these are distinct cells, uniquely accessed by an appropriate structure variable name qualified by a field names. In addition, only field names declared in the structure template can be used to qualify a variable name. And finally, a field name may not be used by itself — it must always qualify an appropriate structure variable. Consider the following examples of structure variable declarations:

```
      struct {
            char f_name[10];
            char m_inits[3];
            char l_name[20];
            int id_no;
            int b_month;
            int b_day;
            int b_year;
      } person, manager;

      struct {
            int id_no;
            float cost;
            float price;
      } part;
```

Here we have declared two variables, `person` and `manager`, to be structures with seven fields, some integers, some strings. In this case, two separate instances of the template are allocated, so person.id_no and manager.id_no are distinct storage cells. We have also defined a variable, `part`, whose template also has a `id_no` field name. But this is also a distinct storage location accessed by `part.id_no`. However, with these declarations, it is NOT legal to refer to the `cost` field of `person` (`person.cost`) or the _day of a `part` (`part.b_day`). Similarly, referring to `f_name` or `price` is

not legal without a variable name of the appropriate type to be qualified. Here are some legal examples of structure usage:

```
part.id_no = 99;
part.cost = .2239;
if (strcmp(person.f_name, "Helen") == 0)
        printf("Last name is %s\n", person.l_name);
printf("This is the cost %d\n",part.cost);
part.price = part.cost * 2.0;
```

The only legal operations allowed on a structure variable are finding the address of the memory block using &, accessing its members, and copying or assigning it as a unit as long as the variables are of an identical structure type, for example:

```
manager = person;
```

12.1.2 Using Structure Tags

As we said above, declaring a structure variable requires two things — describing the template for the structure, and declaring variables of that structure type. It is also possible to perform these two steps in separate statements in a program. That is declare just a structure template with a tag without any variables declared; and later, declare variables of the structure type identified by the tag. For example, this declaration:

```
struct stdtype {
  char name[26];
  int id_number;
};
```

specifies a template with a tag for a structure type, stdtype. (Observe the semicolon after the declaration for the declaration). Such a declaration does NOT allocate memory, since no variables are declared; it merely defines a template for variables declared later. Within the scope of the tag declaration, we can then declare stdtype structure variables like:

```
struct stdtype x, y, z;
```

This declaration allocates memory for three variables, x, y and z, of type structure stdtype; i.e. fitting the template defined earlier. Some additional examples of structure tag and variable declarations are:

```
/* named structure template, no variables declared */
struct date {
    int month;
    int day;
    int year;
};

/* named structure template and a variable declared */
```

```
struct stu_rec {
     char name[30];
     char class_id[3];
     int test[3];
     int project;
     int grade;
} student;

struct stu_rec ee_stu, me_stu;
struct date today, birth_day;
```

The main advantage of splitting the template definition from variable declaration is that the template need be defined only once and may then be used to declare variables in many places. We will see the utility of this below when we pass structures to function.

In general, then, a structure declaration has the following syntax:

struct [<tag_identifier>] {
<type_specifier> <identifier>;
<type_specifier> <identifier>;
...
} [<identifier>[, <identifier>...]];

where the <tag_identifier> and variable identifiers are optional. Once a template has been defined, additional variables of the structure type may be declared by:

struct <tag_identifier> <identifier>[, <identifier>...];

The types of the fields of the structure may be any valid C type; a scalar data type (int, float, etc.), an array, or even a structure. This means that nested structure types can also be declared:

```
struct inventory {
     int item_no;
     float cost;
     float price;
     struct date buy_date;
};

struct car_type{
     struct inventory part;
     struct date ship_date;
     int shipment;
} car;
```

Here, the ship_date field of the car_type structure is itself a date structure (from above) and the part field is an inventory structure, with item_no, cost, etc. fields, including yet another date structure within. The members for nested structures may be accessed with dot operators applied successively from left to right (the grouping for the dot operator is from left to right). Thus:

```
car.ship_date.month = 5;        /* Lvalue is (car.ship_date).month */
car.part.buy_date.month = 12;
```

these assignments refer to the `month` field of the `ship_date` field of the variable `car` and the `month` field of the `buy_date` field of the `part` field of the variable, `car`, respectively.

Both the variables of structure type and the structure tags are frequently referred to as structures. Thus, we may refer to `date` as a structure, and we may say that the variable, `today` is a structure. It is usually clear from the context whether a structure tag or a variable of structure type is meant. However, for the most part, we will use the term structure for the templates themselves, i.e. tags; and, we will specify variables to be of structure type. Thus, `date` is a structure; and `today` is a structure variable or a variable of structure type, i.e. of type, `struct date`.

As with other data types, structures can be initialized in declarations by specifying constant values for the structure members within braces. The initializers for structure members are separated by commas as for an array. For example, a `struct inventory` item can be declared:

```
struct inventory part = { 123, 10.00, 13.50 };
```

which initializes member, `part_no` to 123, member `cost` to 10.00, and member `price` to 13.50. As another example, a `label` item can be declared as:

```
struct name {
        char f_name[10];
        char m_inits[3];
        char l_name[20];
};

struct address {
        char street[30];
        char city[15];
        char state[15];
        int zip};
};

struct label {
        struct name name;
        struct address address;
};

struct label person = { {"Jones", "John", "Paul"},
                {"23 Dole Street", "Honolulu", "Hawaii", 96822} };
```

The structure, `label` has two members, each of which is a structure. The first member, `name`, has three members, and the second member, `address`, has four members. Initialization for each member structure is nested appropriately.

12.1.3 Structures and Functions

Structure variables may be passed as arguments and returned from functions just like scalar variables. Let us consider an example that reads and prints a data record for a part. The record

consists of the part number, its cost and retail price. (In a later section, we will see how an inventory for a list of parts can be maintained). The code to read and print a single part structure is shown in Figure 12.3. Notice we have declared the structure template, `inventory`, at the head of the source file. This is called an *external* declaration and the scope is the entire file after the declaration. Since all the functions in the file use this structure tag, the template must be visible to each of them. The driver calls `read_part()` to read data into a structure and return it to be assigned to the variable, `item`. Next, it calls `print_part()` passing `item` which merely prints the values of the structure fields, one at a time. The program is straightforward. A sample session is shown below:

```
***Part Inventory Data***

Part Number:  2341
Cost:  12.5
Price:  15
Part no.  = 2341, Cost = 12.50, Retail price = 15.00
```

External declarations of structure templates and prototypes facilitate consistent usage of tags and functions. As a general practice, we will declare structure templates externally, usually at the head of the source file. Sometimes, external structure tag declarations will be placed in a separate header file, which is then made part of the source file by an include directive.

From this example, we can see that using structures with functions is no different than using any scalar data type like `int`. However, let us consider what really happens when the program runs. When the function `read_part()` is called, memory is allocated for all of its local variables, including the `struct inventory` variable, `part`. As each data item is read, it is placed in the corresponding field of `part`, accessed with the **dot** operator. The **value** of `part` is then returned to `main()` where it is assigned to the variable `item`. As would be the case for a scalar data type, the value of the return expression is **copied** back to the calling function. Since this is a structure, the entire structure (each of the fields) is copied. For our `inventory` structure, this isn't too bad — only two floats and an integer. If the structure where much larger, maybe including nested structures and arrays, many values would need to be copied.

Likewise with the call to `print_part()`. Here, an `inventory` structure is passed to the function. Recall that in C, all parameters are passed by value — the value of each argument expression is **copied** from the calling function into the cell allocated for the parameter of the called function. Again, for large structures, this may not be a very efficient way to pass data to functions. In the next section we see a way to remedy this problem.

12.1.4   Pointers to Structures

As we saw in the last section, passing and returning structures to functions may not be efficient, particularly if the structure is large. We can eliminate this excessive data movement by passing pointers to the structures to the function, and access them indirectly through the pointers. Figure 12.4 shows a modified version of our previous program which uses pointers instead of passing entire structures.

The code is very similar to Figure 12.3, but we have changed the prototypes and functions to work with pointers. The argument of `read_part()` is a pointer to the `inventory` structure, `item`

```
/*    File: part.c
      This program reads and prints inventory data for a part.
*/
#include <stdio.h>

struct inventory {
      int part_no;
      float cost;
      float price;
};

struct inventory read_part(void);
void print_part(struct inventory part);

main()
{     struct inventory item;

      printf("***Part Inventory Data***\n\n");
      item = read_part();
      print_part(item);
}

/*    Prints data for a single part. */
void print_part(struct inventory part)
{
      printf("Part no. = %d,  Cost = %5.2f, Retail price = %5.2f\n",
                  part.part_no, part.cost, part.price);
}

/*    Reads data for a single part  structure and returns the
      structure.
*/
struct inventory read_part(void)
{     int n;
      float x;
      struct inventory part;
      printf("Part Number: ");
      scanf("%d", &n);
      part.part_no = n;
      printf("Cost: ");
      scanf("%f", &x);
      part.cost = x;
      printf("Price: ");
      scanf("%f", &x);
      part.price = x;
      return part;
}
```

Figure 12.3: Code for Reading and Printing a Single Part

```
/*    File: part.c
      This program reads and prints inventory data for a part.
*/
#include <stdio.h>

struct inventory {
      int part_no;
      float cost;
      float price;
};

void read_part(struct inventory * partptr);
void print_part(struct inventory * partptr);

main()
{     struct inventory item;

      printf("***Part Inventory Data***\n\n");
      read_part(&item);
      print_part(&item);
}

/*    Prints data for a single part pointed to by partptr. */
void print_part(struct inventory * partptr)
{
      printf("Part no. = %d,  Cost = %5.2f, Retail price = %5.2f\n",
                  (* partptr).part_no, (* partptr).cost, (* partptr).price);
}

/*    Reads data for a single part into an object pointed to
      by partptr.
*/
void read_part(struct inventory * partptr)
{     int n;
      float x;
      struct inventory part;
      printf("Part Number: ");
      scanf("%d", &n);
      (* partptr).part_no = n;
      printf("Cost: ");
      scanf("%f", &x);
      (* partptr).cost = x;
      printf("Price: ");
      scanf("%f", &x);
      (* partptr).price = x;
}
```

Figure 12.4: Code for Reading and Printing a Part Using Pointers

declared in `main()`. The function accesses the object pointed to by `partptr`, and uses the dot operator to access a member of that object. Since `partptr` points to an object of type `struct inventory`, we dereference the pointer to access the members of the object:

```
(*partptr).part_no
(*partptr).cost
(*partptr).price
```

Similar changes have been made to `print_part()`. Note, the parentheses are necessary here because the `.` operator has higher precedence than the indirection operator, `*`. We must first dereference the pointer, and then select its appropriate member.

Since, for efficiency, pointers to structures are often passed to functions, and, within those functions, the members of the structures are accessed, the operation of dereferencing a structure pointer and a selecting a member is very common in programs. Therefore, C provides a special pointer operator, $->$, (called *arrow*) to access a member of a structure pointed to by a pointer variable. The operator is a minus symbol, `-`, followed by a greater-than symbol, `>`. This operator is exactly equivalent to a dereference operation followed by the `.` operator as shown below:

$$
\begin{aligned}
\texttt{partptr->part\_no} &\iff \texttt{(*partptr).part\_no} \\
\texttt{partptr->cost} &\iff \texttt{(*partptr).cost} \\
\texttt{partptr->retail} &\iff \texttt{(*partptr).price}
\end{aligned}
$$

The left hand expressions are equivalent ways of writing expressions on the right hand side, e.g. `prtptr->`*member* accesses the member of an object pointed to by `partptr`. Our code for `read_part()` could use the following alternative expressions:

```
partptr->part_no = n;
partptr->cost = x;
partptr->price = x;
```

The general syntax for using the arrow operator is:

<variable_identifier> $->$ <member_identifier>

which is equivalent to:

(* <variable_identifier>).<member_identifier>

We now consider an example using nested structures. The program reads and prints data for a single label consisting of members that are themselves structures. The first member is a structure for a name, the second is a structure for an address. This program is organized in several source and header files as shown in Figure 12.5. (We intend to use the functions in these files for other programs as well).

The driver calls the function `readlabel()` to read in the label data, and the function `printlabel()` to print the label data. Like the previous example, in both function calls, we assume that a pointer to a `struct label` variable is passed as an argument. In the functions, we will use the pointer operator, $->$, to access the members of the object pointed to by the pointer. The function prototypes are shown in the header file `lblutil.h`. The functions are shown in Figure 12.6

The formal parameter in the functions `readlabel()` and `printlabel()` are both a pointer, called `pptr`, which points to an object of type `struct label`. Each function accesses the `first` field of the `name` field of the object pointed to by `pptr` as follows:

```
/*    File: lbl.h
      This file contains structure tags for labels. Label has two members,
      name and address, each of which is a structure type.
*/
struct name_recd {
      char last[15];
      char first[15];
      char middle[15];
};

struct addr_recd {
      char street[25];
      char city[15];
      char state[15];
      long zip;
};

struct label {
      struct name_recd name;
      struct addr_recd address;
};

/* File: lblutil.h */
void printlabel(struct label * personptr);
int readlabel(struct label * personptr);

/*    File: lbl.c
      Other Source Files: lblutil.c
      Header Files: lbl.h, lblutil.h
      This program reads and prints data for one label.
*/
#include <stdio.h>
#include "lbl.h"
#include "lblutil.h"
main()
{     struct label person;

      printf("***Label Data for a Person***\n\n");
      readlabel(&person);
      printf("\nLabel Data:\n");
      printlabel(&person);
}
```

Figure 12.5: Driver and Header Files for Label Program

```
/*   File: lblutil.c */
#include <stdio.h>
#include "lbl.h"
#include "lblutil.h"
#define FALSE 0
#define TRUE 1

/* This routine prints the label data. */
void printlabel(struct label * pptr)
{
    printf("\n%s %s %s\n%s\n%s %s %ld\n",
                pptr->name.first,
                pptr->name.middle,
                pptr->name.last,
                pptr->address.street,
                pptr->address.city,
                pptr->address.state,
                pptr->address.zip);
}

/* This routine reads the label data. */
int readlabel(struct label * pptr)
{   int x;

    printf("Enter Name <First Middle Last>, EOF to quit: ");
    x = scanf("%s %s %s%*c",pptr->name.first,
                    pptr->name.middle,
                    pptr->name.last);
    if (x == EOF)
        return FALSE;
    printf("Enter Street Address: ");
    gets(pptr->address.street);
    printf("Enter City State Zip: ");
    scanf("%s %s %ld%*c",pptr->address.city,
                pptr->address.state,
                &(pptr->address.zip));
    return TRUE;
}
```

Figure 12.6: Code for Label Utility Functions

```
    pptr->name.first
```

Remember, this is the same as:

```
    (*pptr).name.first
```

which means `pptr` is first dereferenced; the `name` field of the dereferenced object is accessed next, and finally the `first` field of `name` is accessed (the dot operator groups from left to right). Similarly, other members of the object pointed to by `pptr` are accessed by:

```
    pptr->name.middle
    pptr->name.last
    pptr->address.street
    pptr->address.city
    pptr->address.state
    pptr->address.zip
```

All the above members, except `zip`, are strings. In `readlabel()`, `scanf()` expects to be passed pointers to objects to store the data read. Since all the string members are already pointers, we need to use the address operator only when we pass the pointer to `pptr->address.zip`. Notice, we use the suppression conversion, `%*c`, to discard the newline character at the end of each line. Thus, after the name is read, `gets()` reads the street address correctly. The function returns `TRUE` if a label is read successfully, and `FALSE` otherwise, i.e. when an EOF is entered by the user for the name, indicating that no label data is available. The function `printlabel()` could have been passed the structure variable itself since it merely needs to print the values of the members; however, as we discussed above, passing a pointer avoids the expense of copying the entire structure. Here is a sample session:

```
    ***Label Data for a Person***

    Enter Name <First Middle Last>, EOF to quit:   John Paul Jones
    Enter Street Address:   23 Dole Street
    Enter City State Zip:   Honolulu Hawaii 96822

    Label Data:

    John Paul Jones
    23 Dole Street
    Honolulu Hawaii 96822
```

12.2   Arrays of Structures

The inventory and the label program examples of the last section handle only a single record. More realistically, a useful program may need to handle many such records. As in previous cases where we needed to store a list of items, we can use an array as our data structure. In this case, the elements of the array are structures of a specified type. For example:

	part_no	cost	price
table[0]—>	123	10.00	15.00
table[1]—>	.	.	.
table[2]—>	.	.	.
table[3]—>	.	.	.

Figure 12.7: A Table of Part Records

```
struct inventory {
    int part_no;
    float cost;
    float price;
};


struct inventory table[4];
```

which defines an array with four elements, each of which is of type **struct inventory**, i.e. each is an **inventory** structure.

We can think of such a data structure as a tabular representation of our data base of parts inventory with each row representing a part, and each column representing information about that part, i.e. the **part_no**, **cost**, and **price**, as shown in Figure 12.7. This is very similar to a two dimensional array, except that in an array, all data items must be of the same type, where an array of structures consists of columns, each of which may be of a distinct data type. As with any array, the array name used by itself in an expression is a pointer to the entire array of structures. Therefore, the following are equivalent ways of accessing the elements of the array.

$$*(\text{table}) \qquad \text{table}[0]$$
$$(\text{table} + 1) \quad \text{table}[1]$$
$$(\text{table} + 2) \quad \text{table}[2]$$
$$(\text{table} + 3) \quad \text{table}[3]$$

With this in mind, let us extend out address label program from Section 12.1.4 to read and print a list of labels. The code is shown in Figure 12.8 and uses the same structures and functions used in program **lbl.c** included in files **lbl.h** and **lblutil.c**.

In the program, the reading of labels is still performed by **readlabel()** only now in a while loop. The loop terminates when either **MAX** number of labels have been read or **readlabel()** returns **FALSE** at end of file. In this case, a new label is not read, but the value of **n** is incremented anyway by the **++** operator. Thus, if the loop is terminated because of an end of file, the incremented value of **n** must be decremented to correctly indicate the number of entries in the array. Finally, labels are printed using **printlabel()** in a loop.

Sample Session:

```
***Labels - Input/Output***
```

```
/*    File: labels.c
      Other Source Files: lblutil.c
      Header Files: lbl.h, lblutil.h
      This program reads in a set of labels, and prints them out.
*/
#define MAX 100
#include <stdio.h>
#include "lbl.h"          /* declarations for the structures */
#include "lblutil.h"      /* prototypes for routines in file lblutil.c */
main()
{     struct label person[MAX];
      int i, n;

      printf("***Labels - Input/Output***\n\n");
      n = 0;
      /* read the labels */
      while (n < MAX && readlabel(&person[n++]))
            ;
      if (n == MAX)
            printf("Labels full - printing labels\n");
      else --n;        /* EOF encountered for last value of n */

      /* print out the labels */
      printf("\nLabel Data:\n");
      for (i = 0; i < n; i++)
            printlabel(&person[i]);
}
```

Figure 12.8: Driver for Address Label Program

```
Enter Name <First Middle Last>, EOF to quit:    John Paul Jones
Enter Street Address:   23 Dole Street
Enter City State Zip:   Honolulu Hawaii 96822
Enter Name <First Middle Last>, EOF to quit:    David Charles Smith
Enter Street Address:   52 University Ave
Enter City State Zip:   Honolulu Hawaii 96826
Enter Name <First Middle Last>, EOF to quit:    ^D


Label Data:


John Paul Jones
23 Dole Street
Honolulu Hawaii 96822


David Charles Smith
52 University Ave
Honolulu Hawaii 96826
```

Next, let us revise the payroll program so that a payroll data record is stored in a structure called **payrecord**. Let us also define a type called **payrecord** for the structure data type that houses a payroll data record:

```
typedef struct payrecord payrecord;
```

We may, thus, declare variables of type **payrecord** rather than **struct payrecord**. The name for the structure tag and the defined data type can be the same as shown. The structure definitions and typedef are placed in the file **payrec.h** and shown in Figure 12.9.

The program logic is simple enough — it reads input data, calculates payroll data, and prints payroll data as before. In this implementation, we will also include calculation of tax withheld. The result is that we have gross pay, net pay, and tax withheld as additional items in payroll data records as seen in the structure definitions. The program also keeps track of totals for gross and net pay disbursed as well as for taxes withheld. The totals are printed as a summary statement for the payroll. Figure 12.10 shows the main driver.

The function **readrecords()** reads the input data records into an array and returns the number of records read, **printrecords()** prints all payroll data records, and **printsummary()** prints the totals of gross pay and taxes withheld. Finally, we need **calcrecords()** to calculate pay for each of the records. These functions are shown in Figures 12.11 and 12.12.

In the code, we use functions **readname()** and **printname()** to read and print an individual name for each record. Finally, we must write **calcrecords()** which calculates the pay for each data record and the totals of gross pay and tax withheld. The tax is calculated on the following basis:

If the total pay is $500 or less, the tax is 15%;

If the total is $1000 or less, the tax is 28%;

If the total is over $1000, the tax is 33%.

```
/*    File: payrec.h */
/*    This file contains structures and data type definitions needed for
      the program in file payrec.c.
*/
struct namerecd {
    char last[15];
    char first[15];
    char middle[15];
};

struct payrecord {
    int id;
    struct namerecd name;
    float hours;
    float rate;
    float regular;
    float overtime;
    float gross;
    float tax_withheld;
    float net;
};

typedef struct payrecord payrecord;
```

Figure 12.9: Data Structure Definitions for Payroll Program

```
/*   File: payrec.c
     Header Files: payrec.h
     This program computes payroll and prints it. Each data record is
     a structure, and the payroll is an array of structures. Tax is
     withheld 15% if weekly pay is below 500, 28% if pay is below 1000,
     and 33% otherwise. A summary report prints out the total gross
     pay and tax withheld.
*/
#include <stdio.h>
#include "payrec.h"
#define MAX 10

void printsummary(double gross, double tax);
int readrecords(payrecord payroll[], int lim);
void printrecords(payrecord payroll[], int n);
double calcrecords(payrecord payroll[], int n, double * taxptr);

main()
{    int i, n = 0;
     payrecord payroll[MAX];
     double gross, tax = 0;

     printf("***Payroll Program***\n\n");
     n = readrecords(payroll, MAX);
     gross = calcrecords(payroll, n, &tax);
     printrecords(payroll, n);
     printsummary(gross, tax);
}
```

Figure 12.10: Driver for Payroll Program

```c
/* File: payrec.c - continued */
/* Function prints total gross pay and total tax withheld. */
void printsummary(double gross, double tax)
{
   printf("\n***SUMMARY***\n\n");
   printf("TOTAL GROSS PAY = $%8.2f; TOTAL TAX WITHHELD = $%8.2f\n",
          gross, tax);
}


/* Function reads payroll input data records until EOF or until lim
records have been read.
*/
int readrecords(payrecord payroll[], int lim)
{  int i, n, x;
   float z;
   void readname(payrecord payroll[], int i);

   for (i = 0; i < lim; i++) {
      printf("Id Number/EOF: ");
      x = scanf("%d%*c", &n);
      if (x == EOF) return i;
      payroll[i].id = n;
      readname(payroll, i);
      printf("Hours Worked: ");
      x = scanf("%f%*c", &z);
      payroll[i].hours = z;
      printf("Rate of Pay: ");
      x = scanf("%f%*c", &z);
      payroll[i].rate = z;
   }
   return i;
}


/* Function reads a single name. */
void readname(payrecord payroll[], int i)
{
   printf("Last Name: ");
   gets(payroll[i].name.last);
   printf("First Name: ");
   gets(payroll[i].name.first);
   printf("Middle Name: ");
   gets(payroll[i].name.middle);
}
```

Figure 12.11: Code for Payroll Program Functions

```
/* Prints a single name. */
void printname(payrecord payroll[], int i)
{
   printf("Name: %s %s %s\n", payroll[i].name.first,
                       payroll[i].name.middle,
                       payroll[i].name.last);
}

/* Function prints n payroll records. */
void printrecords(payrecord payroll[], int n)
{  int i, x;
   float z;
   void printname(payrecord payroll[], int i);

   printf("\n***PAYROLL REPORT***\n\n");
   for (i = 0; i < n; i++) {
      printf("\nId Number: %d\n", payroll[i].id);
      printname(payroll, i);
      printf("Hours Worked: %8.2f  ", payroll[i].hours);
      printf("Rate of Pay: %8.2f\n", payroll[i].rate);
      printf("PAY\n");
      printf("Regular: %8.2f, Overtime = %8.2f, ",
               payroll[i].regular, payroll[i].overtime);
      printf("Gross = %8.2f, Net = %8.2f\n",
               payroll[i].gross, payroll[i].net);
      printf("TAX Withheld = %8.2f\n", payroll[i].tax_withheld);
   }
}
```

Figure 12.12: Code for Payroll Program Functions — continued

The function also keeps a cumulative sum of total gross pay and total tax withheld. Finally, it returns total gross pay and indirectly returns the total tax withheld. The code is shown in Figure 12.13. Here is a sample interaction with the program:

```
***Payroll Program***

Id Number/EOF: 17
Last Name:  Young
First Name:  Cyrus
Middle Name:  Lee
Hours Worked:  38
Rate of Pay:  12
Id Number/EOF: 10
Last Name:  Jones
First Name:  John
Middle Name:  Paul
Hours Worked:  50
Rate of Pay:  16.5
Id Number/EOF: ^D

***PAYROLL REPORT***

Id Number:  17
Name:  Cyrus Lee Young
Hours Worked:  38.00 Rate of Pay:  12.00
PAY
Regular:  456.00, Overtime = 0.00, Gross = 456.00, Net = 387.60
TAX Withheld = 68.40

Id Number:  10
Name:  John Paul Jones
Hours Worked:  50.00 Rate of Pay:  16.50
PAY
Regular:  660.00, Overtime = 247.50, Gross = 907.50, Net = 653.40
TAX Withheld = 254.10

***SUMMARY***

TOTAL GROSS PAY = $ 1363.50; TOTAL TAX WITHHELD = $ 322.50
```

12.3   Sorting Arrays of Structures

We can make one more small improvement to our address label program. Often when we want to print labels, we would like to print them in some sorted order. In this section we will write a function to sort the array of `label` structures. As we saw in Chapter 10, an array is sorted by some *key*, that is, for an array of structures, by a specific member of the structure. A list of

```
/* File: payrec.c - continued */
/* This function computes regular and overtime pay, and the tax to be
   withheld. Tax withheld is 15% of gross pay if not over $500, 28% of
   gross if not over $1000, and 33% of gross otherwise. The function also
   cumulatively sums total gross pay and total tax withheld.
*/
double calcrecords(payrecord payroll[], int n, double * taxptr)
{   int i;
    double gross = 0;
    *taxptr = 0;

    for (i = 0; i < n; i++) {
        if (payroll[i].hours <= 40) {
            payroll[i].regular = payroll[i].gross =
                payroll[i].hours * payroll[i].rate;
            payroll[i].overtime = 0;
        }
        else {
            payroll[i].regular = 40 * payroll[i].rate;
            payroll[i].overtime = (payroll[i].hours - 40) * 1.5 *
                               payroll[i].rate;
        }
        payroll[i].gross = payroll[i].regular + payroll[i].overtime;
        if (payroll[i].gross <= 500)
            payroll[i].tax_withheld = 0.15 * payroll[i].gross;
        else if (payroll[i].gross <= 1000)
            payroll[i].tax_withheld = 0.28 * payroll[i].gross;
        else
            payroll[i].tax_withheld = 0.33 * payroll[i].gross;
        gross += payroll[i].gross;
        *taxptr += payroll[i].tax_withheld;
        payroll[i].net = payroll[i].gross - payroll[i].tax_withheld;
    }
    return gross;
}
```

Figure 12.13: Code for `calcrecords()`

```
/* File: lblutil.c - continued */
/* Sorts an array of labels person[], of size n, by last name
   using an array of pointers plabel[]. */
void sortlabels(struct label person[], struct label *plabel[], int n)
{    int i;

     for (i = 0; i < n; i++)
          plabel[i] = person + i;
     sortptrs(plabel, n);
}


/* Sorts pointers to labels by last name */
void sortptrs(struct label *plabel[], int n)
{    int j, maxpos, eff_size;
     struct label *ptemp;

     for (eff_size = n; eff_size > 1; eff_size--) {
          maxpos = 0;
          for (j = 0; j < eff_size; j++)
               if (strcmp(plabel[j]->name.last,
                        plabel[maxpos]->name.last) > 0)
                    maxpos = j;
          ptemp = plabel[maxpos];
          plabel[maxpos] = plabel[eff_size-1];
          plabel[eff_size-1] = ptemp;
     }
}
```

Figure 12.14: Utility Functions to Sort `label` Structures

labels may be sorted either by last name, or by zip code, or by street address, and so forth. Again, considering the sorting algorithms in Chapter 10, we saw that sorting involves swapping data items to place them in the correct order. However, like passing structures to functions, swapping entire structures can be inefficient if the structures are large. In addition, it is common that an array of structures needs to be sorted by different keys for different purposes. To solve these problems, we can use a technique we used in Chapter 9 for sorting two dimensional arrays — sorting the data using an array of pointers. In this way, the swapping operations while sorting involve only pointers, not entire records, and we can maintain several such pointer arrays, each sorted by a different key.

Figure 12.14 shows the code for the function `sortlabels()` added to the file `lblutil.c` which sorts labels by last name using pointers. (The function assumes the `label` structure defined in `lbl.h` — Section 12.1.4). The function `sortlabels()` is passed the array of labels, `person[]` and an array of pointers to `label` structures, `plabel[]`. This array should be declared in `main()` as:

```
struct label *plabel[MAX];
```

and passed to `sortlabels()` in the call:

```
sortlabels(person,plabel,n);
```

after the `person[]` array is read. The function begins by initializing the elements of `plabel[]` to point to successive elements of the array of structures, `person[]`. It then calls `sortptrs()` to sort the array by last name using these pointers using a selection sort algorithm. The only thing to note is that for the comparison step of the sort, a structure element is accessed by:

```
plabel[j]->name.last
```

which accesses the `last` field of the `name` field of the object pointed to by `plabel[j]`. In the swap step of the sort algorithm, only the pointers are swapped.

We can now write a function, `printsortedlabels()`, to print the labels in sorted order using the `plabel[]` array, modifying `main()` appropriately. We leave this as an exercise.

The utility functions in the file `lblutil.c` provide most of the tools needed to write a useful, interactive address label data base program. In the next chapter, we discuss the remaining piece — file storage for the data base, and write the entire application.

12.4   Unions

In some applications, we might want to maintain information of one of two alternate forms. For example, suppose, we wish to store information about a person, and the person may be identified **either** by name or by an identification number, but never both at the same time. We could define a structure which has both an integer field and a string field; however, it seems wasteful to allocate memory for both fields. (This is particularly important if we are maintaining a very large list of persons, such as payroll information for a large company). In addition, we wish to use the same member name to access identity the information for a person.

C provides a data structure which fits our needs in this case called a *union* data type. A union type variable can store objects of different types at different times; however, at any given moment it stores an object of only one of the specified types. The declaration of a union type must specify all the possible different types that may be stored in the variable. The form of such a declaration is similar to declaring a structure template. For example, we can declare a union variable, `person`, with two members, a string and an integer. If the name is entered, we will use person to store the string; if an identification number is entered, we will use person to store an integer. Here is the union declaration:

```
union {
    int id;
    char name[25];
} person;
```

This declaration differs from a structure in that, when memory is allocated for the variable `person`, only enough memory is allocated to accommodate the **largest** of the specified types. The memory allocated for `person` will be large enough to store the larger of an integer or an 25 character array. Like structures, we can define a tag for the union, so the union template may be later referenced by name:

```
union human {
     int id;
     char name[25];
} person;
```

Likewise, it is possible to declare just a tag, and later, use the tag to declare variables:

```
union human {
     int id;
     char name[25];
};
union human person, *ppers;
```

The syntax for declaring a union type is basically the same as for a structure:

union [<tag_identifier>] {
<type_specifier> <identifier>;
<type_specifier> <identifier>;
...
} [<identifier>[, <identifier>...]];

The members of a union variable may be accessed in the same manner as are members of a structure variable:

<union_var>.<member>
<ptr_to_union_var> − ><member>

Examples include:

```
ppers = &person;
person.id = 12;
if (ppers->id == 12)
     ...
printf("Id = %d\n", person.id);
```

The type of data accessed is determined by the member name used to qualify the variable name. In our example, `person.id` will access an integer; while `person.name` will access a string (a character pointer).

Since at any given time, the contents of the union variable may be one of several types (`int` or string for `person`), we must keep track what type of data is stored in order to access the information correctly. Each time an object is stored in a union variable, it is the programmer's responsibility to keep track of the type stored. If an attempt is made to retrieve a type different from the type last stored, the result is sure to be strange and incorrect. The specific behavior is implementation dependent.

To remember the type of object last stored in a union variable, it is common to store `that` information in a variable. The best way is to declare a structure containing both the union variable as a field and another field that indicates the type of data stored in the union. For example, we can declare such a structure type and a structure array as follows:

```
/*   File: uniutil.c */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "unidef.h"
#include "uniutil.h"
/*   Reads a list of items. Each item is either a string
     or an integer.
*/
int readlist(struct record list[], int lim)
{    int i;
     char s[SIZE];

     printf("Type Identifications For Persons on the List\n");
     printf("Either a Name or an Id Number, EOF to quit\n");
     for (i = 0; i < lim && gets(s); i++) {
          if (isdigit(*s)) {                    /* Is it a number? */
               list[i].ptype = INT;             /* If so, store type, */
               list[i].person.id = atoi(s);     /* and the ID number. */
          }
          else {                                /* Otherwise, */
               list[i].ptype = NAME;            /* Store string type, */
               strcpy(list[i].person.name, s); /* and the NAME. */
          }
     }
     return i;                                  /* Return no. of items. */
}
```

Figure 12.15: Reading Data into a Union Variable

```
#define NAME 0
#define ID 1
struct record {
     int ptype;
     union human person;
};
struct record list[MAX];
```

Now, as we read information about each element of list, if the information is numeric, we store it as id; otherwise, we store it as name. We also store the type, ID or NAME in the member, ptype.

Figure 12.15 shows a function that reads identifying information about each person and stores it in the union type member. Depending on the type of information read, it uses the appropriate union field name, and stores the type in the ptype field of the structure. The loop body in the function looks at the first character of the input string, s. If it is a digit, then the data is an id number so INT is stored in ptype, and the string is converted to an integer (using atoi()) and

```
/*    File: uniutil.c - continued */
/*    Prints out the list of items. Each item is either a string
      or an integer.
*/
void printlist(struct record list[], int n)
{    int i;

     printf("Identifications of People on the List\n");
     printf("Either a Name or an ID Number\n");
     for (i = 0; i < n; i++) {
          if (list[i].ptype == INT)
               printf("Id number: %d\n", list[i].person.id);
          else
               printf("Name: %s\n", list[i].person.name);
     }
}
```

Figure 12.16: Printing Information from a Union Variable

stored in the union id field. If the first character of s is not a digit, NAME is stored in ptype, and the string is copied into the union name field.

It is now easy to write a function that prints the identifying information stored in the list. Since each record includes the type of information stored in the union variable, it is easy to retrieve the information correctly as shown in Figure 12.16.

We now write a simple program that first reads a list of identifying information about a group of people, and later prints the list. The identifying information may be either a name or an id number. The structure and union declarations as well as constant definitions are included in the file unidef.h shown together with the code in Figure 12.16.

Sample Session:

```
***Union Variables - Lists***

Type Identifications For Persons on the List
Either a Name or an Id Number, EOF to quit
John Kent
345
Jane Ching
231
Mary Smith
^D
Identifications of People on the List
Either a Name or an ID Number
Name:   John Kent
Id number:   345
Name:   Jane Ching
```

```
/*   File: unidef.h */
#define INT 0
#define NAME 1
#define MAX 10
#define SIZE 25
union human {
     int id;
     char name[SIZE];
};
struct record {
     int ptype;
     union human person;
};


/*   File: uniutil.h */
int readlist(struct record list[], int lim);
void printlist(struct record list[], int n);


/*   File: union.c
     Other Source Files: uniutil.c
     Header Files: unidef.h, uniutil.h
     This program illustrates the use of union variables. It reads
     a list of items identifying people either by name or by id
     number. It then prints out the list. Each item is stored in
     a union variable either as a name or as an integer. The list
     is kept in an array of structure record. The structure record
     has two members, the union variable and a variable that stores
     the type of object stored in the union.
*/
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "unidef.h"
#include "uniutil.h"
main()
{    struct record list[MAX];
     int n;

     printf("***Union Variables - Lists***\n\n");
     n = readlist(list, MAX);
     printlist(list, n);
}
```

Figure 12.17: Header File and Driver Program for Union Example

```
Id number:   231
Name:  Mary Smith
```

The above program can be written in many alternate ways. We have written the program to illustrate the use of union variables.

12.5   Common Errors

Common errors occur when pointers are used to reference structures and their members. It is best to use parentheses around dereferenced pointers, `(*p).member`, or to use the operator, `->`, e.g. `p->member`, when referencing a member of a structure pointed to by a pointer.

12.6   Summary

In this chapter, we have described the last remaining data types provided by the C language: structures and unions. A structure allows the grouping of various pieces of related information of different types into one variable. It is declared by defining a *template* specifying the type of each data item stored in the structure and giving each *member* or *field* a name:

```
struct [<tag_identifier>] {
<type_specifier> <identifier>;
<type_specifier> <identifier>;
...
} [<identifier>[, <identifier>...]];
```

Variables may be declared when the template is defined or, if a *tag* is used to name the template, may be declared later using the tag:

```
struct <tag_identifier> <identifier>[, <identifier>...];
```

which allocates storage for all members. Fields of a structure variable may be accessed using the "dot" (.) operator:

```
<variable_identifier>.<member_identifier>
```

called *qualifying* the variable name. Such qualified structure variable expressions may be used like the corresponding field type in a program. Structure variables may be passed to and returned from functions, but it is more common to use pointers to structures to avoid excessive copying. Members of a structure can be accessed with a pointer using the $->$ operator:

```
<variable_identifier> − ><member_identifier>
```

which is equivalent to:

```
(* <variable_identifier>).<member_identifier>
```

We have illustrated the use of structures with various programming examples.

Finally, we have described the union data type, which is defined similar to structures; however, has the semantics of only one of the member types being resident in such a variable at one time. That is, a union allows several different types of information to be stored in the same physical space at different times. For a union variable, storage is allocated only for the largest of the data types which may reside in the variable.

Structures are a valuable tool for developing complex programs and data structures in an efficient and top down manner.

12.7   Exercises

1. Find and correct errors if any. What will be the output?

```
struct node {
     int id;
     int score;
}

#include <stdio.h>
main()
{    struct node *px, x, y;
     px = &x;
     while (scanf("%d %d", px.id, px.score) != EOF)
          printf("%d   %d\n", *px.id, *px.score);
}
```

2. Define a data structure, `intflt`, that will allow one to store either an integer or a float. Read strings and convert them to either integers or floats depending on whether there is a fractional part present. Store the resulting values in an `intflt` type array. When the input is terminated, print the stored values.

12.8   Problems

1. Write the function `printsortedlabels()` described in Section 12.3 and make the modifications to `main()` to read a list of address labels and print them in sorted order by last name.

2. Define a structure with the following members:

   ```
   social security number
   id number
   name (last, first, middle)
   exam score
   ```

   Use the above structure for the data record of one student in a class of 50 students maximum. Write a menu-driven program that allows the commands: read data from an input file into an array of the above structure; print data on screen; save data into an output file; sort the data by a specified primary key using pointers to the array; quit.

3. Modify Problem 2 to allow more than one exam up to a maximum of 5 exams. Use an array of exam scores in the structure. Assume that the first three lines of the input file include course title and headings. The actual data starts with the fourth line.

4. Modify Problem 3 to compute and store a weighted average of the exam scores for each student. Weighted average should be a member of the structure. Also allow for computation of an average of any one or all the exams.

5. Modify Problem 4 to allow deleting one or more records, changing one or more records, adding one or more records.

6. Modify Problem 5 so that it can read an input file which may or may not contain a column for the weighted average. Allow the user to output the data but specify which data fields are not to be output to a new file.

7. Modify the above program to include scores for a number of projects up to a maximum of 15. Weighted average must now include exam as well as project scores. Allow a structure member for a letter grade.

8. Modify the above program so it allows the user to perform the following functions: form a class grade list for a new class; enter grades for a project or an exam; change grades for a project or an exam; add or delete a student from a class list; calculate the average for a project or an exam; calculate the weighted average for each student over the projects and the exams; sort the data by a primary key, e.g. weighted average, exam2, proj3, etc.; sort the data by a primary key and a secondary key, i.e. if two records have the same primary key, then sort those records by a secondary key; plot a distribution of the weighted average grades.

9. Write a program that keeps a membership list for a private club. The data fields required are:

```
name
spouse name if any
address, business, residence
telephone, business, residence
hobby interests
membership date
dues outstanding
other charges outstanding
```

The club has a limit of 100 members. Write a program that allows the club manager to: maintain the club list and update it; send out a mailing list to all members with all data about the club members, except for financial data; send out reminders to members about the charges outstanding; post new charges and dues at regular intervals; post paid amounts upon receipt.

10. Assume that the above club maintains a library of at most 500 books. Data for each book consists of:

```
book number
title
author
co-authors
publisher
date published
subject
keywords
check out data:
     name, address, phone
     date checked out
     data returned
     charges, if any
```

Write a program to maintain the library including: search the library by book number, author, title, subject, keywords; add new books, remove outdated books (all books older than 5 years); check out books; late charges at $0.25 per day if a book is out by more than a month; write data to a file for books overdue and charges.

11. Write a macro processor assuming that the macros do not have arguments. Use a structure to keep a macro identifier and its replacement string. Read an input file which may have macros, and create an output file with macros replaced by replacement strings.

12. Write a macro processor. Assume that macros may have arguments. Use structures to keep data about a macro.

13. Use a structure to represent a rational number. Write functions for rational number arithmetic. Write a simple calculator program for rational numbers.

Chapter 13

Files and the Operating System

So far, our interaction with the Operating System of the computer has been limited to using the compiler and shell to create executable files from our programs and execute them, as well as limited interaction with the File System to provide input data to our programs and store the output of results. All of our file I/O, either redirected standard input and output or direct using library functions such as `fscanf()` and `fprintf()`, has been with ASCII files using the formatted I/O utilities provided in C. In this chapter we look at an alternate method of doing I/O — block I/O, where a binary image of a data structure can be stored or retrieved. We discuss the library routines for performing block I/O and managing access to such files. We then provide an example program, a small data base system, which makes use of this facility.

Finally, we discuss other facilities provided by the C library for interacting with the shell from within a program, such as executing a shell command and command line arguments for our program.

13.1   Block Input/Output

The file I/O functions discussed so far perform reading and writing of different types of data using *formatted* ASCII information stored in files. Each I/O operation acts on a stream of character bytes, and the appropriate conversions from characters to an internal representation is performed by the library routines. While it is convenient to have data stored in files in an ASCII form (such data can be read or written by other programs and devices such as text editors, printers, etc.), it can be tedious and inefficient to perform all that data conversion from ASCII to internal binary, and back to ASCII); particularly for structure type data with many members of different types.

C provides additional file I/O library functions which allow direct input or output of the binary, internal representation of data to files. This form of I/O is called *block I/O*, because data is transferred in blocks directly from the file to storage locations in memory with no conversion. It should be noted that the files that store such data are **binary** files and cannot be read or written directly by other operating system programs such as text editors or printer software. Only a program which knows the organization of binary information within the files can access them correctly.

The library functions provided for this type of I/O are `fwrite()` and `fread()`. The function `fwrite()` writes (i.e. appends) a block of data of specified size to a file. Similarly, `fread()` reads a block of data of specified size into a memory location. The prototypes for these functions are

defined in `stdio.h` and they may be described as:

fread   *Prototype:*   `size_t fread(void *buffer,`   *in:* `<stdio.h>`
                              `size_t size, size_t no_items,`
                              `FILE *fp);`

               Returns: actual number of items read (may be less than `no_items`; `NULL` if error or
               end of file.

               Description: This function reads `no_items`, each of size, `size`, bytes from stream, `fp`,
               into `buffer`.

fwrite   *Prototype:*   `int fwrite(const void *buffer, int size,`   *in:* `<stdio.h>`
                              `int no_objs, FILE *stream);`

               Returns: the number of objects written if successful; less than `no_objs` on error.

               Description: This function writes (appends) `no_objs` objects of size, `size`, from
               `buffer` to `stream`.

The function prototypes use the data type, `size_t`, defined in `stdio.h`, which is of an unsigned type. (This is the type actually returned by the `sizeof operator`. As stated above, `fread()` returns the number of items read. When an end of file is encountered before `no_items` items are read, the return value will be less than the number requested; so this may be used to indicate end of file. Also note that the first parameter of the prototype for `fwrite()` uses a `const` qualifier. This ensures that no attempt is made to change the object pointed to by `buffer` within the function. Many prototypes use `const` qualifiers in the parameter declarations to prevent unplanned changes.

To see how to use these functions, let us use them to copy a file. The program logic is straight forward: open the input and output files, read each block of characters from the input file into a buffer, write each block to the output file. When a short size block is read, terminate the loop, write the short block, and close the files. The code is shown in Figure 13.1. We have declared a buffer of type `signed char` so it can store any arbitrary bytes. We have also declared a pointer, `ptr`, with the qualifier `const`, since it is to remain unchanged, and initialized it to point to the buffer, `buf`. After the files are opened, the while loop reads a block of 100 items of `char` size from the input file, and writes the block to the output file. The loop is terminated when less than 100 items are read, indicating less than 100 items were remaining in the file to be read, so the end of file has been reached. The number of items read are assigned to n. At this point, the number of data bytes in the buffer is stored in `n`, so writing a block of 100 items would result in some garbage output. (The data from the previous block is still present in the rest of the buffer). Instead, the final block of `n` items is written.

This program simply copies blocks of 100 bytes at a time from the input file to the output file. These files can be any type of file, such as text files, program files, other ASCII files, even binary files, at least on Unix systems. However, on some non-Unix systems, the system routines may not be able to read or write arbitrary binary information unless the mode strings passed to `fopen()` explicitly indicates opening a file for binary I/O by appending the character 'b' to the string. Thus, the mode strings must be `"rb"` or `"wb"` instead of `"r"` or `"w"`. A program using block I/O to copy binary files on an IBM PC using DOS operating system and a TURBO C compiler is shown in Figure 13.2. The program is the same as before except for the mode strings used in function calls to `fopen()`.

```
/*    File: blkcopy.c
      The program uses block I/O to copy a file.
*/
#include <stdio.h>
main()
{    signed char buf[100];
     const void *ptr = (void *) buf;
     FILE *input, *output;
     size_t n;

     printf("***File Copy - Block I/O***\n\n");
     printf("Input File: ");
     gets(buf);
     input = fopen(buf, "r");
     if (!input) {
          printf("Unable to open input file\n");
          exit(0);
     }
     printf("Output File: ");
     gets(buf);
     output = fopen(buf, "w");
     if (!output) {
          printf("Unable to open output file\n");
          exit(0);
     }
     while ((n = fread(ptr, sizeof(char), 100, input)) == 100)
          fwrite(ptr, sizeof(char), 100, output);
     fwrite(ptr, sizeof(char), n, output);
     close(input);
     close(output);
}
```

Figure 13.1: Copying a File Using Block I/O

```
/*    File: bincopy.c
      This program copies a binary file. Standard files are not allowed.
*/
#include <stdio.h>

main()
{     int c;
      char s[25];
      FILE *input, *output;

      printf("***Binary File Copy - Character I/O***\n\n");
      printf("Input File: ");
      gets(s);
      input = fopen(s, "rb");

      if (!input) {
            printf("Unable to open input file\n");
            exit(0);
      }

      printf("Output File: ");
      gets(s);
      output = fopen(s, "wb");

      if (!output) {
            printf("Unable to open output file\n");
            exit(0);
      }

      while ((c = fgetc(input)) != EOF)
            fputc(c, output);
      close(input);
      close(output);
}
```

Figure 13.2: File Copy Program for Binary Files

13.2 Tell and Seek

The file functions described in the last section, as well as the formatted file I/O functions, read and write data sequentially from the *current position* in the file. This position in the stream is one of the pieces of information stored in the FILE data structure which we get a pointer to when the file is opened. The standard library provides a function to determine the current position, ftell(). The position returned from ftell() is in terms of the number of bytes offset from the beginning of the file. The library also provides a function which allows a program to move the current position elsewhere in stream: fseek().

ftell   *Prototype:* `long ftell(FILE *stream);`   *in:* `<stdio.h>`

Returns: the current file pointer in stream measured in bytes from the beginning. If there is an error, it returns -1L.

fseek   *Prototype:* `int fseek(FILE *stream, long offset,`   *in:* `<stdio.h>`
                     `int base);`

Returns: 0 if successful; nonzero on failure. (DOS returns zero even if pointer is not repositioned.)

Description: The function, fseek, sets the file pointer for stream to a new position that is offset bytes from the location specified by base. For binary files, offset is the number of bytes to offset. For text files, offset must be zero or a value returned by ftell(). The value of base must be 0 to indicate the beginning of the file, 1 for the current position, or 2 for the end of file. stdio.h defines three constants for the base position: SEEK_SET is 0, SEEK_CUR is 1, and SEEK_END is 2.

Let us illustrate the use of the above functions in a simple program. Suppose we wish to write a program that opens a file and begins to read items from the file. Suppose at some point, the program requires the size of the file. Then, after the size is determined, the program should resume reading items from where it left off.

The program will: read and print a number of strings from a file; then, call a function filesize() that returns the size of the file; and finally, resume reading and printing strings from the file. The program driver is shown in Figure 13.3.

Let us now write filesize(), which returns the size of the file, fp. It determines the size of the file by moving the current position to the end of the file and finding the offset of the pointer in bytes. However, before the current position is moved to the end of the file, the function must save the position where it left off. This saved position is restored by filesize() before it returns to the calling program. The function is shown in Figure 13.4.

The function first saves the current position in savepos; then moves the pointer to the end of the file. It uses 0L for offset since the offset must be long. Next, the function uses ftell() to find the offset in bytes of the current position, which is now at the end of the file. The value of end is, therefore, the size of the file. Finally, the saved position is restored and the file size is returned. Here is a sample session:

```
***File Seek - File Size***

File Name:  payin.dat
```

```
/*    File: seek.c
      This program illustrates the use of fseek() to reposition
      a file pointer. The program reads a specified number of strings
      from a file and prints them. Then, the program calls on filesize()
      to print out the size of the file. After that the program resumes
      reading and printing strings from the file.
*/

#include <stdio.h>
#define MAX 81
long int filesize(FILE *fp);

main()
{     int m, n = 0;
      FILE *fp;
      char s[MAX];

      printf("***File Seek - File Size***\n\n");
      printf("File Name: ");
      gets(s);
      fp = fopen(s, "r");

      if (!fp)
          exit(0);
      printf("Number of lines in first printing: ");
      scanf("%d", &m);

      while (fgets(s, MAX, fp)) {   /* read strings */
          fputs(s, stdout);         /* print the strings */
          n++;
          if (n == m)    /* if m string are printed, print file size */
              printf("Size of file = %ld\n", filesize(fp));
      }

      fclose(fp);
}
```

Figure 13.3: Driver for Program Illustrating `ftell()` and `fseek()`

```
/*   File: seek.c - continued */
/*   Returns the size of the file stream fp.*/
long int filesize(FILE *fp)
{    long int savepos, end;

     savepos = ftell(fp);      /* save the file pointer position */
     fseek(fp, 0L, SEEK_END); /* move to the end of file */
     end = ftell(fp);          /* find the file pointer position */

     fseek(fp, savepos, SEEK_SET); /* return to the saved position */
     return end;                    /* return file size */
}
```

Figure 13.4: Code for `filesize()`

```
Number of lines in first printing:  3
/* File:  payin.dat */
ID Last First Middle Hours Rate
------------------------------------------------------------------------
Size of file = 238
5 Jones Mike David 40 10
7 Johnson Charles Ewing 50 12
12 Smythe Marie Jeanne 35 10
```

In the sample session, the first three lines of `payin.dat` are printed and then the size of the file is printed as 238 bytes. Finally, the rest of the file `payin.dat` is printed.

A few comments on the use of `fseek()`:

For text files, the offset value passed to `fseek()` can be either 0 or a value returned by `ftell()`.

When `fseek()` is used for binary files, the offset must be in terms of actual bytes.

13.3   A Small Data Base Example

A data base is a collection of a large set of data. We have seen several examples of data bases in previous chapters, such as our payroll data and the list of address labels discussed in Chapter 12. In our programs working with these data bases we have simply read data from files, possibly performed some calculations, and printed reports. However, to be a useful data base program, it should also perform other management and maintenance operations on the data. Such operations include editing the information stored in the data base to incorporate changes, saving the current information in the data base, loading an existing data base, searching the data base for an item, printing a report based on the data base, and so forth. Programs that manage data bases can become quite elaborate, and such a program to manage a large and complex data base is called a *Data Base Management System (DBMS)*.

```
/*    File: lbldb.h
      This file contains structure tags for labels. Label has two
      members, name and address, each of which is a structure type.
*/
struct name_recd {
      char last[15];
      char first[15];
      char middle[15];
};

struct addr_recd {
      char street[25];
      char city[15];
      char state[15];
      long zip;
};

struct label {
      struct name_recd name;
      struct addr_recd address;
};

typedef struct label label;
```

Figure 13.5: Data Structure Definitions for Label Data Base Program

In this section we will implement a rather small data base system that maintains our data base for address labels. We will assume that there are separate lists of labels for different groups of people; therefore, it should be possible to save one list in a file named by the user as well as to load a list from any of these files. The data in a list of labels is mostly fixed; however, it should be possible to make additions and/or changes. It should also be possible to sort and search a list of labels.

In our skeleton data base system, we will not implement sorting and searching operations (we have already implemented a sort function for labels in Section 12.3), instead, our purpose here is to illustrate some of the other operations to see the overall structure of a DBMS. We will implement operations to add new labels, print a list of labels, as well as loading and saving lists in files. The program driver will be menu driven. The user selects one of the items in the menu, and the program carries out an appropriate task. The data structures we will use include the **label** structure and a type, **label**, defined in the file **lbldb.h** shown in Figure 13.5. The program driver is shown in Figure 13.6.

The list of labels is stored in the array, **lbllist[]**, and **n** stores the actual number of labels, initially zero. A new list is read by the function **load()** which returns the number of labels loaded. A list can be edited by **edit()** which updates the value of **n**. Both **edit()** and **load()** must not exceed the maximum size of the array. The functions **print()** and **save()** write **n** labels from the

```
/*   File: lbldb.c
     Header Files: lbldb.h
     This program initiates a data base for labels. It allows the
     user to edit labels, i.e., add new labels, save labels in a
     file, load a previously saved file, and print labels.
*/

#define MAX 100
#include <stdio.h>
#include <ctype.h>
#include "lbldb.h"          /* declarations for the structures */

main()
{    char s[25];
     label lbllist[MAX];
     int n = 0;

     printf("***Labels - Data Base***\n");
     printf("\nCommand: E)dit, L)oad, P)rint, S)ave, Q)uit\n");
     while (gets(s)) {

          switch(toupper(*s)) {
               case 'E': n = edit(lbllist, n, MAX); break;
               case 'L': n = load(lbllist, MAX); break;
               case 'P': print(lbllist, n); break;
               case 'S': save(lbllist, n); break;
               case 'Q': exit(0);
               default: printf("Invalid command - retype\n");
          }
          printf("\nCommand: E)dit, L)oad, P)rint, S)ave, Q)uit\n");
     }
}
```

Figure 13.6: Driver for Label Data Base Program

current list.

Figure 13.7 shows a partial implementation of `edit()`, allowing only the addition of new labels. It does not implement operations for deletion or change of a label. The `edit()` function presents a sub-menu and calls the appropriate function to perform the task selected by the user. We have included program "stubs" for the functions `del_label()` and `change_label()` which are not yet implemented. The `add_label()` function calls on `readlbl()` to read one label. If a label is read by `readlbl()` it returns `TRUE`; otherwise, it returns `FALSE`. The loop that reads labels terminates when either the maximum limit is reached or `readlbl()` returns `FALSE`. Each time `readlbl()` is called, `n` is updated, and the updated value of `n` is returned by `add_label()`. In turn, `edit()` returns this value of `n` to the main driver.

The function `readlbl()` first reads the last name, as shown in Figure 13.8. If the user enters an empty string, no new label is read and the function returns `FALSE`; otherwise, the remaining informations for a label is read and the function returns `TRUE`.

The `print()` function calls on `printlabel()` to print a single label data to the standard output. The functions are shown in Figure 13.9.

Finally, we are ready to write functions `load()` and `save()`. We will use `fread()` and `fwrite()` to read or write a number of structure items directly from or to a binary file. This method of storing the data base is much more efficient that reading ASCII data, field by field for each label. The code is shown in Figure 13.10. The function `load()` opens an input file, and uses `fread()` to read the maximum possible (`lim`) items of the size of a `label` from the input file. The buffer pointer passed to `fread()` is the pointer to the array of `labels`, `lbllist`. Finally, `load()` closes the input file and returns `n`, the number of items read. Similarly, `save()` opens the output file, and saves `n` items of `label` size from the buffer to the output file. It then closes the output file and returns `n`. If it is unable to open the specified output file, it returns 0. A sample session is shown below.

```
***Labels - Data Base***

Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
l
Input File:  lbl.db

Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
p

Label Data:

James Edward Jones
25 Dole St
Honolulu Hi 96822

Jane Mary Darcy
23 University Ave
Honolulu Hi 96826

Helen Gill Douglas
```

```
/*   File: lbldb.c - continued */
/*   Edits labels: adding labels has been implemented so far. */
int edit(label lbllist[], int n, int lim)
{    char s[80];

     printf("A)dd, D)elete, C)hange\n");
     gets(s);

     switch(toupper(*s)) {
          case 'A': n = add_label(lbllist, n, lim);
                    break;
          case 'D': del_label();
                    break;
          case 'C': change_label();
                    break;
          default: ;
     }

     return n;
}


/*   Adds new labels to lbllist[] which has n labels. The maximum
     number of labels is lim.
*/
int add_label(label lbllist[], int n, int lim)
{
     while (n < lim && readlbl(&lbllist[n++]))
          ;

     if (n == lim)
          printf("Maximum number of labels reached\n");
     else --n;        /* EOF encountered for last value of n */

     return n;
}

void del_label(void)
{
   printf("Delete Label not yet implemented\n");
}

void change_label(void)
{
   printf("Change Label not yet implemented\n");
}
```

Figure 13.7: Partial Code for Editing the Data Base

```
/*    File: lbldb.c - continued */
/*    Includes and defines included at the head of the file. */
#define FALSE 0
#define TRUE 1

/*    This routine reads the label data until the name is a blank. */
int readlbl(struct label * pptr)
{     int x;
      char s[25];

      printf("Enter Last Name, RETURN to quit: ");
      gets(s);
      if (!*s)
            return FALSE;
      else strcpy(pptr->name.last, s);
      printf("Enter First and Middle Name: ");
      x = scanf("%s %s%*c",pptr->name.first, pptr->name.middle);
      printf("Enter Street Address: ");
      gets(pptr->address.street);
      printf("Enter City State Zip: ");
      scanf("%s %s %ld%*c",pptr->address.city, pptr->address.state,
              &(pptr->address.zip));
      return TRUE;
}
```

Figure 13.8: Code for `readllbl()`

```
/*    File: lbldb.c - continued */
/*    Prints n labels stored in lbllist[]. */
void print(label lbllist[], int n)
{    int i;

     printf("\nLabel Data:\n");
     for (i = 0; i < n; i++)
          printlabel(&lbllist[i]);
}


/* This routine prints the label data. */
void printlabel(struct label * pptr)
{
     printf("\n%s %s %s\n%s\n%s %s %ld\n",
               pptr->name.first,
               pptr->name.middle,
               pptr->name.last,
               pptr->address.street,
               pptr->address.city,
               pptr->address.state,
               pptr->address.zip);
}
```

Figure 13.9: Code for `print()` and `printlabel()`

```
/*   File: lbldb.c - continued */
/*   Loads a maximum of lim labels from a file into lbllist[].
     Returns the number n of labels actually read.
*/
int load(label lbllist[], int lim)
{    char s[25];
     FILE *infp;
     int n;

     printf("Input File: ");
     gets(s);
     infp = fopen(s, "r");
     if (!infp)
          return 0;
     n = fread(lbllist, sizeof(label), lim, infp);
     fclose(infp);
     return n;
}

/*   Saves n labels from lbllist[] to a file. */
int save(label lbllist[], int n)
{    char s[25];
     FILE *outfp;

     printf("Output File: ");
     gets(s);
     outfp = fopen(s, "w");
     if (!outfp)
          return 0;
     fwrite(lbllist, sizeof(label), n, outfp);
     fclose(outfp);
     return n;
}
```

Figure 13.10: Code for load() and save

```
123 Kailani Ave
Kailua Hi 96812

Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
```
e
```
A)dd, D)elete, C)hange
```
a

```
Enter Last Name, RETURN to quit:
```
Springer
```
Enter First and Middle Name:
```
John Karl
```
Enter Street Address:
```
Coconut Ave
```
Enter City State Zip:
```
Honolulu Hi 96826
```
Enter Last Name, RETURN to quit:

Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
```
s
```
Output File:
```
lbl.db

```
Command:  E)dit, L)oad, P)rint, S)ave, Q)uit
```
q

The session starts with the menu menu. We select the menu item Load to load a previously saved list of labels in the file, `lbl.db`. After this file is loaded, we select Print to print the labels. Next, we select Edit and Add to add one new label. Then we select Save to save the revised list to the file `lbl.db`. Finally, we select Quit to exit the program.

13.4 Operating System Interface

As stated at the beginning of the chapter, all of our programs so far have had minimal interaction with the environment in which they are running, i.e. the operating system, and in particular the shell. One area where we could make use of operating system support is in specifying files to be used in execution of the program. In our previous examples we have either redirected the input or output when running the program (and read or written to the standard input or output in the program code), or prompted the user explicitly for the file names once the program has begun executing. However, this is not the only (nor most convenient) way to specify files to a program. It should also be possible to pass arguments to a program when it is executed. An executable program is invoked by a command to the host operating system consisting of the name of the program. However, the entire command may also include any arguments that are to be passed to the program. For example, the C compiler does not prompt us for the file names to be compiled; instead we simply type the command:

```
cc filename.c
```

The entire command is called the *command line* and may include additional information to the program such as *options* and file names. The C compiler (and most, if not all, other commands) is also simply a C program.

There must be a way this additional information can be passed to an executing program. There is. The command line arguments are passed to the formal parameters of the function `main()`. We have always defined the function `main()` with no formal parameters; however, in reality it does have such parameters. The formal parameters of `main()` are: an integer, `argc`, and an array of pointers, `argv[]`. The full prototype for `main()` is:

```
int main(int argc, char * argv[]);
```

Each word typed by the user on the command line is considered an argument (including the program name). The parameter `argc` receives the integer count of the number of arguments on the command line, and each word of the line is stored in a string pointed to by the elements of `argv[]`. So, if the command line consists of just the program name, `argc` is 1 and `argv[0]` points to a string containing the program name. If there are other arguments on the command line, `argv[1]` points to a string containing the first argument after the program name, `argv[2]` to the next following one, and so forth. In addition, `main()` has an integer return value passing information back to the environment specified by the `return` or `exit` statement terminating `main()`. Recall, we have always used `exit` in the form:

```
exit(0);
```

A common convention in Unix is that a program terminates with a zero return value if it terminates normally, and with non-zero if it terminates abnormally.

Figure 13.11 shows a program that prints the values of `argc` and each of the strings pointed to by the array `argv[]`. The program then uses the first argument passed on the command line as a "source" file name, and the second as a "destination" file name and copies the source file to the destination. The program returns zero to the environment to indicate normal termination.

Sample Session with a command line:

```
filecopy filecopy.c xyz

    ***Command Line Arguments - File Copy***

    Number of arguments, argc = 3
    The arguments are the following strings
    C:\BK\BOOK\CH9\FILECOPY.EXE
    filecopy.c
    xyz
```

The number of arguments in the command line is 3, and each of the strings pointed to by the array `argv[]` is then printed. The first argument is the complete path for the program name as interpreted by the host environment. The program then opens the files and copies the file `filecopy.c` to `xyz`.

In addition to receiving information from the operating system, a program can also call on the shell to execute commands available in the host environment. This is very simple to do with C using the library function `system()`. Its prototype is:

```
int system(const char *cmmdstr);
```

```
/*    File: filecopy.c
      This program shows the use of command line arguments. argc is the number
      of words in the command line. The first word is the program name, the next
      is the first argument, and so on. The program copies one file to another.
      The command line to copy file1 to file2 is:

            filecopy file1 file2
*/

#include <stdio.h>
main(int argc, char *argv[])
{     int i, c;
      FILE *fin, *fout;

      printf("***Command Line Arguments - File Copy***\n\n");
      printf("Number of arguments, argc = %d\n", argc);
      printf("The arguments are the following strings\n");

      /* argv[0] is the program name, */
      /* argv[1] is the first argument after the program name, etc. */
      for (i = 0; i < argc; i++)
          printf("%s\n", argv[i]);

      fin = fopen(argv[1], "r");
      fout = fopen(argv[2], "w");

      if( !fin || !fout ) exit(1);
      while ((c = fgetc(fin)) != EOF)
          fputc(c, fout);
      exit(0);
}
```

Figure 13.11: File Copy Program Using Command Line Arguments

The function executes the command given by the string `cmmdstr`. it returns 0 if successful, and returns -1 upon failure. Examples include:

```
system("date");
system("time");
system("clear");
```

The first prints the current date, the second prints the current time maintained by the system, and the third clears the screen.

13.5   Summary

In this chapter we have looked at alternate file I/O functions, `fread()` and `fwrite()` which perform *block* I/O; transferring blocks of data directly between memory and data files. This form of I/O is more efficient than *formatted* I/O which converts information between its internal binary representation and the corresponding ASCII representation of the information as strings for the actual I/O. It should be remembered that files used for block I/O have information stored in **binary** and are therefore NOT readable by other programs which do not know the format of the data.

We also saw library routines for controlling the "current position" in the file stream for I/O; namely `ftell()` and `fseek()`. These operations can be performed on either text or binary files.

Finally, we discussed the interactions a program can perform with its environment — the operating system or shell. These include receiving information from the shell in the form of command line arguments which are passed to `main()` as arguments, and the `system()` function which can call on the environment to perform some command.

13.6   Problems

1. Write a program that copies one file to another with file names supplied by the command line.

2. Modify the program in Problem 8 in Chapter 12 to add load and store operations to the student data base program using block I/O.

3. Modify the program in Problem 9 in Chapter 12 to add load and store operations to the club data base program using block I/O.

4. Modify the program in Problem 10 in Chapter 12 to add load and store operations to the library data base program using block I/O.

5. Write a program that serves as a dictionary and thesaurus. A dictionary keeps a meaning for each word. A meaning may be one or more lines of text. A thesaurus keeps a set of synonyms for each word. Assume that the maximum number of entries in the dictionary is 500; there are no more than two lines for a meaning; and there are no more than three synonyms for each word. Allow the user to ask for synonyms, meanings, spell check a text file with repacement of words or add word entries to dictionary. Use files to load and save the dictionary.

Chapter 14

Storage Class and Scope

In previous chapters we have discussed the declaration of variables within functions and described how memory space is allocated by the compiler for these variables as a program executes. How (and where) this memory is allocated, as well as how long it is allocated is determined by what is called the *storage class* for the variable. In addition we have discussed where within the code the variable name is "visible", i.e. where it can be accessed by name. This is called the *scope* of the variable. The variables we have seen so far have all been of storage class *automatic*, i.e. they are allocated when the function is called, and deallocated when it returns, with *local* scope, i.e. visible only within the body of the function. The C language provides several other storage classes together with their scope for controlling memory allocation. In this chapter we will discuss in more detail the concepts of memory allocation and present the other storage classes available in C, viz. *automatic*, *external*, *register*, and *static*. We will also see that functions, as well as variables, have storage class and scope. We next discuss *dynamic* allocation of memory, where a program can determine how much additional memory it needs as it executes. Finally, we introduce *function pointers*, i.e. pointer variables which can hold pointers to functions rather than data. We will see how these pointers are created, stored, passed as parameters, and accessed.

14.1 Storage Classes

Every C variable has a storage class and a scope. The storage class determines the part of memory where storage is allocated for an object and how long the storage allocation continues to exist. It also determines the scope which specifies the part of the program over which a variable name is visible, i.e. the variable is accessible by name. The are four storage classes in C are automatic, register, external, and static.

14.1.1 Automatic Variables

We have already discussed automatic variables. They are declared at the start of a block. Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block. The scope of automatic variables is local to the block in which they are declared, including any blocks nested within that block. For these reasons, they are also called *local variables*. No block outside the defining block may have direct access to automatic variables, i.e. by name. Of course, they may be accessed indirectly by other blocks and/or functions using pointers.

```
/* File: reg.c */
main()
{     register float a = 0;
      auto int bb = 1;
      auto char cc = 'w';

      /* rest of the program */
}
```

Figure 14.1: Code fragment illustrating `register` and `auto` declarations

o

Automatic variables may be specified upon declaration to be of storage class `auto`. However, it is not required; by default, storage class within a block is `auto`. Automatic variables declared with initializers are initialized each time the block in which they are declared is entered.

14.1.2   Register Variables

Register variables are a special case of automatic variables. Automatic variables are allocated storage in the memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing in the CPU. These computers often have small amounts of storage within the CPU itself where data can be stored and accessed quickly. These storage cells are called *registers*.

Normally, the compiler determines what data is to be stored in the registers of the CPU at what times. However, the C language provides the storage class `register` so that the programmer can "suggest" to the compiler that particular automatic variables should be allocated to CPU registers, if possible. Thus, `register` variables provide a certain control over efficiency of program execution. Variables which are used repeatedly or whose access times are critical, may be declared to be of storage class `register`.

Register variables behave in every other way just like automatic variables. They are allocated storage upon entry to a block; and the storage is freed when the block is exited. The scope of register variables is local to the block in which they are declared. Rules for initializations for register variables are the same as for automatic variables.

Figure 14.1 shows a code fragment for a `main()` function that uses `register` as well as `auto` storage class. The class specifier simply precedes the type specifier in the declaration. Here, the variable, `a`, should be allocated to a CPU register by the compiler, while `bb` and `cc` will be allocated storage in memory. Note, the use of the `auto` class specifier is optional.

As stated above, the `register` class designation is merely a suggestion to the compiler. Not all implementations will allocate storage in registers for these variables, depending on the number of registers available for the particular computer, or the use of these registers by the compiler. They may be treated just like automatic variables and provided storage in memory.

Finally, even the availability of register storage does not guarantee faster execution of the program. For example, if too many register variables are declared, or there are not enough registers available to store all of them, values in some registers would have to be moved to temporary storage

in memory in order to clear those registers for other variables. Thus, much time may be wasted in moving data back and forth between registers and memory locations. In addition, the use of registers for variable storage may interfere with other uses of registers by the compiler, such as storage of temporary values in expression evaluation. In the end, use of register variables could actually result in slower execution. Register variables should only be used if you have a detailed knowledge of the architecture and compiler for the computer you are using. It is best to check the appropriate manuals if you should need to use register variables.

14.1.3 External Variables

All variables we have seen so far have had limited scope (the block in which they are declared) and limited lifetimes (as for automatic variables). However, in some applications it may be useful to have data which is accessible from within any block and/or which remains in existence for the entire execution of the program. Such variables are called *global variables*, and the C language provides storage classes which can meet these requirements; namely, the external and static classes.

External variables may be declared outside any function block in a source code file the same way any other variable is declared; by specifying its type and name. No storage class specifier is used — the position of the declaration within the file indicates external storage class. Memory for such variables is allocated when the program begins execution, and remains allocated until the program terminates. Fo rmost C implementations, every byte of memory allocated for an external variable is initialized to zero.

The scope of external variables is global, i.e. the entire source code in the file following the declarations. All functions following the declaration may access the external variable by using its name. However, if a local variable having the same name is declared within a function, references to the name access the local variable cell. Figure 14.2 shows an example of external variables and their scope. The comments in the code indicate which variable is accessed in each reference to the name. The situation is shown graphically in Figure 14.3. Executing the program produces the following sample session:

```
***Scope of External Variables***

a1 = 2
a1 = a, b1 = 77
a1 = 2
a1 = 13, b1 = 19.200001
a1 = 13
```

External variables may be initialized in declarations just as automatic variables; however, the initializers must be constant expressions. The initialization is done only once at compile time, i.e. when memory is allocated for the variables variables.

In general, it is a good programming practice to avoid use of external variables as they destroy the concept of a function as a "black box". The black box concept is essential to the development of a modular program with *independent* modules. With an external variable, any function in the program can access and alter the variable, thus making debugging more difficult as well. This is not to say that external variables should *never* be used. There may be occasions when the use of an external variable significantly simplifies the implementation of an algorithm. Suffice it to say that external variables should be used rarely and with caution.

```
/*    File: glb.c
      This program clarifies the scope of external variables.
*/
#include <stdio.h>
void next(void);
void next1(void);

int a1 = 1;          /* external variable:  global scope */
                     /* scope:  main(), next(), next1() */


main()
{
    printf("***Scope of External Variables***\n\n");
    a1 = 2;                      /* external var */
    printf("a1 = %d\n", a1); /* a1 = 2 */
    next();
    printf("a1 = %d\n", a1); /* a1 = 2 */
    next1();
    printf("a1 = %d\n", a1); /* a1 = 13 */
}


int b1 = 0;          /* external variable */
                     /* scope: global to next, next1 */
                     /* main() cannot access b1 */


void next(void)
{   char a1;         /* auto var: scope local to next()  */
                     /* next() cannot access external a1 */
    a1 = 'a';        /* local auto var */
    b1 = 77;         /* external var */
    printf("a1 = %c, b1 = %d\n", a1, b1);   /* a1 = a, b1 = 77 */
}


void next1(void)
{   float b1;        /* auto var: scope local to next1()  */
                     /* next1() cannot access external b1 */
    b1 = 19.2;       /* auto var */
    a1 = 13;         /* external var */
    printf("a1 = %d, b1 = %f\n", a1, b1);   /* a1 = 13 */
                                            /* b1 = 19.2 */
}
```
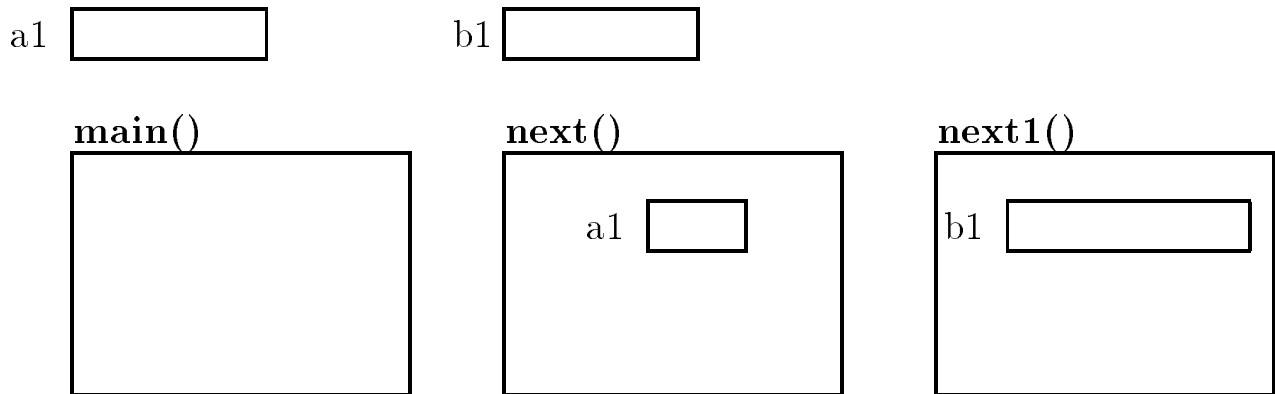
Figure 14.2: Example of external variable scope

Figure 14.3: Storage allocation for global variables

14.1.4 Variable Definition vs Declaration

Up until now, we have been using the term *declaration* rather loosely when referring to variables. In this section, we will "tighten" the definition of this term. So far when we have "declared" a variable, we have meant that we have told the compiler *about* the variable; i.e. its type and its name, as well as allocated a memory cell for the variable (either locally or globally). This latter action of the compiler, allocation of storage, is more properly called the *definition* of the variable. The stricter definition of *declaration* is simply to describe information "about" the variable.

So far, we have used declarations to *declare* variable names and types as well as to *define* memory for them. Most of the time these two actions occur at the same time, that is, most declarations are definitions; however, this may not always be the case.

We have already seen an analogous case illustrating the difference between *declaring* and *defining* with functions. The prototype statement for a function *declares* it, i.e. tells the compiler "about" the function — its name, return type, and number and type of its parameters. A similar statement, the function header, followed by the body of the function, *defines* the function — giving the details of the steps to perform the function operation.

For automatic and register variables, there is no difference between definition and declaration. The process of declaring an automatic or a register variable defines the variable name and allocates appropriate memory. However, for external variables, these two operations may occur independently. This is important because memory for a variable must be allocated only once, to ensure that access to the variable always refers to the same cell. Thus, all variables must be defined once and only once. If an external variable is to be used in a file other than the one in which it is *defined*, a mechanism is needed to "connect" such a use with the uniquely defined external variable cell allocated for it. This process of connecting the references of the same external variable in different files, is called *resolving the references*.

As we saw in the previous section, external variables may be defined and declared with a declaration statement outside any function, with no storage class specifier. Such a declaration allocates memory for the variable. A declaration statement may also be used to simply *declare* a variable name with the `extern` storage class specifier at the beginning of the declaration. Such a declaration specifies that the variable is *defined* elsewhere, i.e. memory for this variable is allocated in another file. Thus, access to an external variable in a file other than the one in which

it is *defined* is possible if it is *declared* with the keyword `extern`; no new memory is allocated. Such a declaration tells the compiler that the variable is defined elsewhere, and the code is compiled with the external variable left unresolved. The reference to the external variable is resolved during the linking process.

Here are some examples of *declarations* of external variables that are not *definitions*:

```
extern char stack[10];
extern int stkptr;
```

These declarations tell the compiler that the variables `stack[]` and `stkptr` are defined elsewhere, usually in some other file. If the keyword `extern` were omitted, the variables would be considered to be new ones and memory would be allocated for them. Remember, access to the same external variable defined in another file is possible only if the keyword `extern` is used in the declaration. Figure 14.4 shows an example of a source program that references the same external variable in different files. The files are assumed to be compiled separately and linked together to create a load module. A sample run is shown below.

```
***Declaration vs Definition***

a1 = 2
a1 = 13
```

14.1.5   An Example: Lexical Scanner

To illustrate the use of external storage class variables, let us now consider an example in which a good program design is facilitated by the use of an external variable. The task is to find the next *token* in an input stream of characters. A *token* is a useful chunk of characters in the input stream, e.g. an operator, an identifier, an integer, a floating point number, etc. Tokens are also called *symbols*. A function that finds the next token in an input stream and identifies its type is called a *lexical scanner*. For our example, we will write a simple lexical scanner, `get_token()`, to find the next token and its type until an end of file is reached.

We will assume that the only valid tokens in the input stream to be identified by the program are either integers or operators. Further, we assume that integers can have no more than five digit characters and the operator can have no more than a single character. The operators allowed are `+`, `-`, `*`, `/`. If an integer type token exceeds the size limit, an oversize type is to be identified. White space characters between tokens are to be ignored. Any other character is an invalid character which is to be identified as an illegal type of token. Finally, the end of file is to be signaled by an end_of_text type of token.

We assume that `get_token()` determines the next token in the input stream and its type. We use a file `symdef.h` for all the defines. The function prototype for `get_token()` is included in `symtok.h`. The function takes two arguments: a string for the token, and the maximum size of the token. The function returns the type of the token, a symbolic constant with an integer value. The files `symdef.h` and `symtok.h` are shown in Figure 14.5. The logic for the driver is straightforward and the implementation is in the file called `symbol.c` shown in Figure 14.6. A loop is executed as long as there is a new token, and for each iteration, a token and its type are printed. When the end of file is reached, the token type returned by `get_token()` is `EOT`, the loop is terminated and

```
/*    File: ext.c
      This example shows reference to an external variable
      in more than one file. The program is organized in
      three files. The external variable a1 is defined in ext.c,
      and it is declared as extern in FILE3.C.
*/
#include <stdio.h>
void next(void);
void next1(void);
int a1 = 1;                     /* definition of external a1 */

main()
{
    printf("***Declaration vs Definition***\n\n");
    a1 = 2;
    printf("a1 = %d\n",a1);         /* a1 = 2 */
    next();                         /* No change in external a1 */
    next1();                        /* external a1 changed to 13 */
    printf("a1 = %d\n", a1);        /* a1 = 13 */
}

/*    File: FILE2.C */
int b1 = 0;          /* definition of external b1 */
void next(void)
{   char a1;         /* auto a1 defined */

    a1 = 'a';        /* only local a1 is visible */
    b1 = 77;         /* external b1 is accessed */
}

/*    File: FILE3.C */
extern int a1;       /* declaration of external a1 */
void next1(void)
{   float b1;        /* auto b1 defined */

    b1 = 19.2;       /* only local b1 is visible */
    a1 = 13;         /* external a1 is accessed */
}
```

Figure 14.4: Example of the use of **extern** declarations

```
/*    File: symdef.h */
      /* Token Types */
      #define INT 0        /* integer */
      #define OPR 1        /* operator */
      #define ILG 2        /* illegal */
      #define EOT 3        /* end of text */
      #define OVR 4        /* oversize */

      #define LIM 5        /* token size limit */

/* File: symtok.h */
int get_token(char * token, int lim);
```

Figure 14.5: Header files for Lexical Scanner

```
/*    File: symbol.c
      Other Source Files: symtok.c, symio.c
      Header Files: symdef.h, symtok.h, symio.h
      This program reads an input stream and determines the tokens in the
      input stream. The primary token types are integer and operator. If
      the integer type token exceeds a specified limit, the token is of
      type oversize. Leading white space is skipped over. All other
      characters are considered to be illegal type tokens. Finally, EOF is
      returned as a special token type to terminate the program.
*/
#include <stdio.h>
#include "symdef.h"
#include "symtok.h"

main()
{     int type;
      char symbol[LIM + 1];

      printf("***Tokens and Types***\n\n");
      printf("Types: integers(0), operators(1), illegal(2),\n");
      printf("       end of text(4), and oversize integers(5)\n");
      printf("Type input text, EOF to quit\n");
      while ((type = get_token(symbol, LIM)) != EOT)
          printf("Token = %8s Type = %8d\n", symbol, type);
}
```

Figure 14.6: Driver for Lexical Scanner

the program ends. The size limit on a token is defined by LIM. The string, symbol, has a size of
LIM plus one to accommodate the terminating NULL character.

Here is our logic for get_token(). The function scans the input stream, skipping over any
leading white space. The first non-white character determines the type of token to build. For
example, if the first non-white character is a digit character, the function builds a token of type
INT. The integer type token is built using a loop. As long as the input character is a digit character
and the token size limit is not exceeded, the input character is appended to the token string. If the
token size limit is exceeded, the type is identified as OVR and the digit is discarded. The process of
discarding digits continues until a non-digit character is read. The token string is terminated with
a NULL, and the token type is returned. Otherwise, the building of an integer token is terminated
when a non-digit character is read. The non-digit character read must somehow be returned to the
input stream, so that it is available in building the next token. For example, if the next character is
an operator, +, that character must be used in building the next token. If this non-digit character
were discarded, it would be lost. Thus, the extra character that was read must be placed back
into the input stream to be available once again for building the next token.

We will assume that the desired I/O actions are performed using an "effective input stream".
We will write two functions, getchr() and ungetchr(c) for I/O with the effective stream. The
function getchr() correctly reads a character from the effective input stream, and ungetchr(c)
puts a character, c, back into the effective input stream. Assuming these functions, the algorithm
for building an integer type token is simple:

```
if (isdigit(c)) {    /* if c is a digit, */
     type = INT;            /* type is integer */
     while (isdigit(c)) {  /* repeat as long as c is a digit: */
          if (i < lim)           /* if the size limit is not exceeded, */
               s[i++] = c;    /* append the digit char; */
          else type = OVR;    /* otherwise, we have an oversize token */
          c = getchr();  /* get the next input char */
     }
     s[i] = NULL;          /* append the NULL */
     ungetchr(c);          /* put back the extra char read */
}
```

The prototypes for the functions getchr() and ungetchr() are:

```
/* File: symio.h */
int getchr(void);
void ungetchr(int c);
```

Assuming these functions are available in the source file, symio.c, we can implement the
function get_token() in Figure 14.7. Finally, we are ready to write the functions getchr() and
ungetchr() in a separate file. We will use a buffer to simulate the effective input stream so that
when a character is to be returned to the input stream, it is placed in the buffer. When a character
is to be read, the buffer is examined first. If there is a character in the buffer, that character is
taken as the next input character. If the buffer is empty, a new character is read from standard
input using getchar(). Thus, the one character buffer serves as an adjunct to the input stream;
getchr() gets the next character either from the buffer or from the standard input, depending on

```c
/*   File: symtok.c */
#include <stdio.h>
#include "symdef.h"
#include "symio.h"
#include <ctype.h>
#define TRUE 1
#define FALSE 0


/*   Gets the next token s with a size limit of lim,
     and returns the token type.
*/
int get_token(char s[], int lim)
{    int i, c, type;

     i = 0;                      /* initialize string index i to zero */
     c = getchr();               /* get the first character */
     while (isspace(c))          /* skip over white space */
          c = getchr();
     if (isdigit(c)) {           /* if c is a digit */
          type = INT;            /* type is INT */
          while (isdigit(c)) {     /* Build an INT token */
               if (i < lim)        /* if size limit not exceeded, */
                    s[i++] = c;    /* add the next char to token; */
               else type = OVR;    /* else, type is OVR */
               c = getchr();       /* get next char */
          }
          ungetchr(c);         /* and put back the extra char read. */
     }
     else if (is_op(c)) {      /* if c is an operator */
          s[i++] = c;          /* build an operator token */
          type = OPR;
     }
     else if (c == EOF)        /* if end of file */
          type = EOT;          /* type is EOT */
     else {
          type = ILG;          /* otherwise, we have an illegal char */
          s[i++] = c;          /* a single char string is built */
     }
     s[i] = NULL;              /* terminate the token string */
     return(type);             /* return token type */
}
/* Checks to see if c is an operator */
int is_op(int c)
{
     if (c == '+' || c == '-' || c == '*' || c == '/')
          return(TRUE);
     return(FALSE);
}
```

Figure 14.7: Code for get_token()

```
/*   File: symio.c */
#include <stdio.h>
#include "symdef.h"       /* needed for stdio.h */

int c = NULL;             /* buffer c initialized to zero */
                          /* initialization unnecessary */


/*   Gets the next character either from the buffer if there is
     one, otherwise gets a char from stdin.
*/
int getchr(void)
{    int ch;
     if (c) {             /* if c is not a null char, */
          ch = c;         /* save it temporarily, and  */
          c = NULL;       /* reset c to NULL */
          return ch;      /* return the saved value */
     }
     else
          return getchar();  /* else, return a char from stdin */
}


/*   Puts a char into the buffer for later use */
void ungetchr(int cc)
{
     c = cc;            /* save the char cc in the buffer */
}
```

Figure 14.8: Code for implementing the "effective input stream"

the state of the buffer, while `ungetchr()` saves a character into the buffer for later use. Effectively, `getchr()` gets a character from the input stream, and `ungetchr()` returns a character to the input stream. Both `getchr()` and `ungetchr()` must access the buffer. However, `get_token()` should not be concerned with the details of accessing the input stream. Such details should be *hidden* from the rest of the program. Such information hiding is an important component of modular program design. The above case obviously calls for it; thus, `get_token()` should not be involved with the details of maintaining the buffer.

To achieve this information hiding, we put `getchr()` and `ungetchr()` in a separate file together with the external variable used as a one character buffer which is accessible to both `getchr()` and `ungetchr()`. Figure 14.8 shows the implementation. The external variable for the character buffer used in the file `symio.c` makes it unnecessary for other functions to pass a buffer variable as an argument in function calls to `getchr()` and `ungetchr()`. Separation of these functions and the external variable they use into a distinct file makes for a modular program design. No other function needs access to the external variable defined in the file `symio.c`.

A standard library function, `ungetch()`, is available which returns its argument to the keyboard

buffer. We could have also used `ungetch()` and `getchar()` to handle the above tasks of getting and ungetting characters from the keyboard input stream.

A sample run of the program `symbol.c` is shown below:

```
***Tokens and Types***

Types:  integers(0), operators(1), illegal(2),
end of text(4), and oversize integers(5)
Type input text, EOF to quit
123 * 723456 + 12
Token = 123 Type = 0
Token = * Type = 1
Token = 72345 Type = 4
Token = + Type = 1
Token = 12 Type = 0
45+23*7
Token = 45 Type = 0
Token = + Type = 1
Token = 23 Type = 0
Token = * Type = 1
Token = 7 Type = 0
x = 8;
Token = x Type = 2
Token = = Type = 2
Token = 8 Type = 0
Token = ; Type = 2
^D
```

In the first input line, we use blanks to separate the tokens. We also have an oversize token in this case. In the second input line, no blanks are used to separate the tokens. Finally, the last line includes many illegal characters. In each case, the longest possible token is built.

While we caution against the use of external variables as a rule, there are occasions when the use of external variables results in better programs. The deciding factor should always be better program design that provides modularity and flexibility, and that facilitates debugging.

14.1.6   Static Variables

As we have seen, external variables have global scope across the entire program (provided `extern` declarations are used is files other than where the variable is defined), and a lifetime over the the entire program run. The storage class, static, similarly provides a lifetime over the entire program, however; it provides a way to limit the scope of such variables, Static storage class is declared with the keyword `static` as the class specifier when the variable is defined. These variables are automatically initialized to zero upon memory allocation just as external variables are. Static storage class can be specified for automatic as well as external variables.

Static automatic variables continue to exist even after the block in which they are defined terminates. Thus, the value of a static variable in a function is retained between repeated function

calls to the same function. The scope of static automatic variables is identical to that of automatic variables, i.e. it is local to the block in which it is defined; however, the storage allocated becomes permanent for the duration of the program. Static variables may be initialized in their declarations; however, the initializers must be constant expressions, and initialization is done only once at compile time when memory is allocated for the static variable.

Figure 14.9 shows an example which sums integers, using static variables. Function `sumit()` reads a new integer and keeps a cumulative sum of the previous value of the sum and the new integer read in. The cumulative value of sum is kept in the static variable, `sum`. The driver, `main()` calls `sumit()` five times to sum five integers.

Sample Session:

```
***Static Variables***

Please enter 5 numbers to be summed
Enter a number:   12
The current total is 12
Enter a number:   23
The current total is 35
Enter a number:   34
The current total is 69
Enter a number:   45
The current total is 114
Enter a number:   56
The current total is 170
Program completed
```

While the static variable, `sum`, would be automatically initialized to zero, it is better to do so explicitly. In any case, the initialization is performed only once at the time of memory allocation by the compiler. The variable `sum` retains its value during program execution. Each time the function `sumit()` is called, `sum` is incremented by the next integer read.

Static storage class designation can also be applied to external variables. The only difference is that static external variables can be accessed as external variables only in the file in which they are defined. No other source file can access static external variables that are defined in another file.

```
/* File: xxx.c */
static int count;
static char name[8];
main()
{
    ... /* program body */
}
```

Only the code in the file `xxx.c` can access the external variables count and name. Other files cannot access them, even with `extern` declarations.

We have seen that external variables should be used with care, and access to them should not be available indiscriminately. Defining external variables to be static provides an additional

```
/*    File: static.c */
/*    Program uses a function to sum integers. The function
      uses a static variable to store the cumulative sum.
*/
#include <stdio.h>
#define MAX 5
void sumit(void);

main()
{    int count;

     printf("***Static Variables***\n\n");
     printf("Please enter 5 numbers to be summed\n");
     for (count = 0; count < MAX; count++)
          sumit();
     printf("Program completed\n");
}


/*    Function reads an integer, and keeps cumulative sum of
      integer read and the previous value of a static variable sum.
*/
void sumit(void)
{    static int sum = 0;        /* sum is initialized to zero  */
                                /* at compile time. */
     int num;

     printf("Enter a number: ");
     scanf("%d",&num);
     sum += num;
     printf("The current total is %d\n",sum);
}
```

Figure 14.9: An example of static variables

```
/*   File: symio2.c */
#include <stdio.h>
#include "symdef.h"

static int c = NULL;      /* static external c */

/*   Gets the next character either from the buffer if there is
     one, otherwise gets a char from stdin.
*/
int getchr()
{  int ch;

     if (c) {             /* if c is not a null char, */
          ch = c;         /* save it temporarily, and  */
          c = NULL;       /* reset c to zero */
          return(ch);     /* Return the saved value */
     }
     else
          return(getchar());  /* else, return a char from stdin */
}


/*   Puts a char into the buffer for later use */
void ungetchr(int cc)
{
     c = cc;             /* save the char cc in the buffer */
}
```

Figure 14.10: Revised file `symio.c` using `static` variable

control on which functions can access them. For example, in the `symbol.c` example in the last section, we created a file `symio.c` which contained an external variable. This external variable should be accessible only to the functions in that file. However, there is no way to guarantee that some other file may not access it by declaring it as **extern**. We can ensure that this will not happen by declaring the variable as **static** as shown in Figure 14.10. The static variable `c` would not be accessible to functions defined in any other file, thus preventing an unplanned use of it as an external variable by the code in other files.

14.1.7   Storage Class for Functions

Like variables, functions in C have a storage class and scope. All functions in C are external by default and are accessible to all source files. However, functions may be declared to be of static class, in which case they are accessible only to functions in the file in which they are defined, not to functions in other files. This is another way of hiding information. Information hiding makes these static function names invisible to all other files; thus, these names may be used to define

other functions elsewhere.

Here is an example that uses static variables as well as a static function. The program assigns bins to different part numbers. The array, `index`, represents the bin number where the part number is stored (it is easy to generalize the program to structures). The program is organized in two files, `bins.c` and `binutil.c`. The first file, `bins.c`, contains the driver which reads in the part numbers, and calls a function, `getbin()`, to assign a bin number to each part number. Finally, the driver prints the bins and the corresponding part numbers using the function `printbin()`. Here are the prototypes:

```
/*   File: binutil.h */
void getbin(int bin[], int part, int lim);
void printbin(int bin[], int lim);
```

The function `getbin()` needs three arguments: an array of bins, a part number, and the array size limit. The bin number is just the array index, so `getbin()` assigns one of the bins in the array to the part number, and stores the part number in the array at the corresponding bin number index. The function `printbin()` needs the array of bins and its size as arguments. It prints out each bin number index and the corresponding part number stored at that array index. The driver is shown in Figure 14.11. The program loop reads a part number and if it is not zero, it calls `getbin()` to assign a bin number to the part number. If the part number is zero, the loop terminates, and `printbin()` prints bin numbers and corresponding part numbers.

Let us now implement `getbin()`. Unused array elements of bin should be initialized to some invalid part number, say -1, so that `printbin()` would be able to distinguish the valid elements of the array. The first time `getbin()` is called, it calls `initbin()` which initializes bin to -1. In addition, `getbin()` should assign the next available index to the part number. The functions are shown in Figure 14.12. The function `getbin()` uses a static variable, `first`, initialized to `TRUE`, to determine if the function is being called for the first time. When the function is called the first time, it initializes the array, `bin` and changes first to `FALSE`. A second static variable, `bin_number` is used to remember the next available bin number between function calls. As a bin is assigned to a part number, `bin_number` is incremented. Since it is a static variable, its latest value is available each time the function is called. The function `printbin()` merely prints each array index and the part number stored at that index. Initialization of the array bin is done by a static function `initbin()`. This function is not required anywhere else, and so a static class is declared for it, thus the details of array initialization are hidden from all other functions. A sample run of the program is shown below:

```
***Bin Assignments to Parts***

Type part numbers, enter zero to quit
Enter part number:   1523
Enter part number:   234
Enter part number:   725
Enter part number:   9120
Enter part number:   0
Bin number 0 has part number 1523
Bin number 1 has part number 234
```

```
/*   File: bins.c
     Other Source Files: binutil.c
     Header Files: binutil.h
     This program assigns a unique bin number to each part number. The
     user types the part numbers and the program assigns bin numbers
     to the parts in sequence. A zero part number terminates the program.
     It is also assumed that the user types only new part numbers. No
     check is made to see if a part number is already assigned a bin.
*/
#include <stdio.h>
#include "binutil.h"      /* prototypes for getbin(), printbin() */
#define MAX 100

main()
{    int bin[MAX], part_no;

     printf("***Bin Assignments to Parts***\n\n");
     printf("Type part numbers, enter zero to quit\n");
     do {
          printf("Enter part number: ");
          scanf("%d",&part_no);
          if (part_no)
               getbin(bin, part_no, MAX);
     } while (part_no);
     printbin(bin, MAX);
}
```

Figure 14.11: Driver for bins program

```
/*   File: binutil.c */
#include <stdio.h>
#include "binutil.h"      /* prototypes for getbin(), printbin() */
#define TRUE 1
#define FALSE 0
static void initbin(int bin[], int lim);

/*   Initializes an array bin of size lim. The function
     is declared static since no other file needs it.
*/
static void initbin(int bin[], int lim)
{    int i;

     for (i = 0; i < lim; i++)
          bin[i] = 0;
}


/*   Assigns a bin element to a part number. First time it
     is called, it initializes the array bin[].
*/
void getbin(int bin[], int part, int lim)
{    static int first = TRUE;
     static bin_number = 0;

     if (first) {
          initbin(bin, lim);
          first = FALSE;
     }
     if (bin_number < lim)
          bin[bin_number++] = part;
     else
          printf("Error - out of Part Bins\n");
}


/*   Prints out bin numbers and part numbers. */
void printbin(int bin[], int lim)
{    int i;

     for (i = 0; i < lim && bin[i]; i++)
          printf("Bin number %d has part number %d\n",i,bin[i]);
}
```
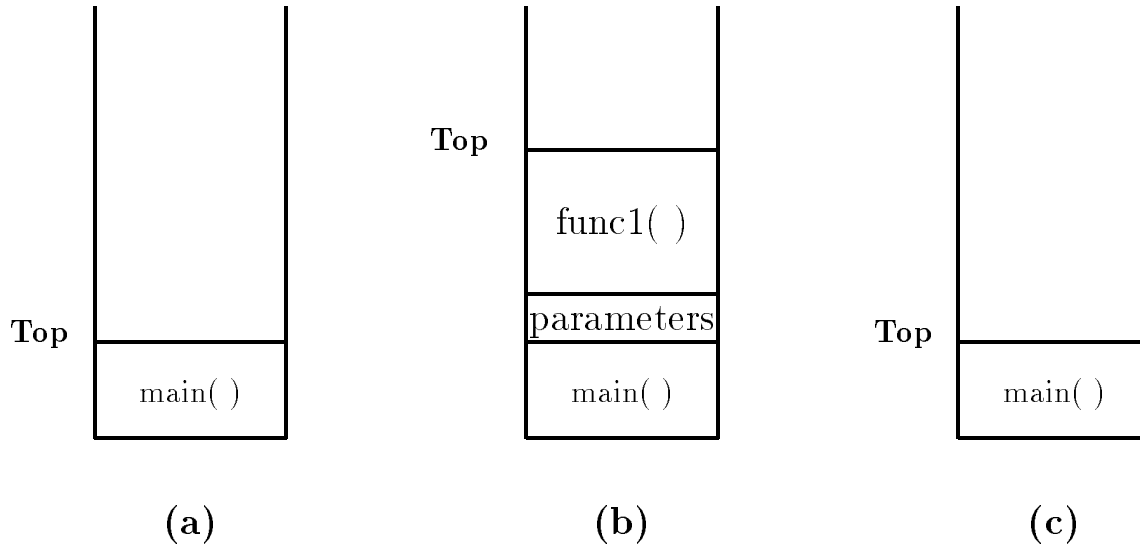
Figure 14.12: Code for bin utilities

Figure 14.13: Organization of the Stack

```
Bin number 2 has part number 725
Bin number 3 has part number 9120
```

14.1.8 Stack vs Heap Allocation

We conclude our discussion of storage class and scope by breifly describing how the memory of the computer is organized for a running program. When a program is loaded into memory, it is organized into three areas of memory, called *segments*: the *text segment*, *stack segment*, and *heap segment*. The text segment (sometimes also called the code segment) is where the compiled code of the program itself resides. This is the machine language representation of the program steps to be carried out, including all functions making up the program, both user defined and system.

The remaining two areas of system memory is where storage may be allocated by the compiler for data storage. The stack is where memory is allocated for automatic variables within functions. A stack is a *First In First Out* (FIFO) storage device where new storage is allocated and deallocated at only one "end", called the Top of the stack. This can be seen in Figure 14.13. When a program begins executing in the function `main()`, space is allocated on the stack for all variables declared within `main()`, as seen in Figure 14.13(a). If `main()` calls a function, `func1()`, additional storage is allocated for the variables in `func1()` at the top of the stack as shown in Figure 14.13(b). Notice that the parameters passed by `main()` to `func1()` are also stored on the stack. If `func1()` were to call any additional functions, storage would be allocated at the new Top of stack as seen in the figure. When `func1()` returns, storage for its local variables is deallocated, and the Top of the stack returns to to position shown in Figure 14.13(c). If `main()` were to call another function, storage would be allocated for that function at the Top shown in the figure. As can be seen, the memory allocated in the stack area is used and reused during program execution. It should be clear that memory allocated in this area will contain garbage values left over from previous usage.

The heap segment provides more stable storage of data for a program; memory allocated in the heap remains in existence for the duration of a program. Therefore, global variables (storage

class external), and static variables are allocated on the heap. The memory allocated in the heap area, if initialized to zero at program start, remains zero until the program makes use of it. Thus, the heap area need not contain garbage.

14.2   Dynamic Memory Allocation

In the previous section we have described the the storage classes which determined how memory for variables are allocated by the compiler. When a variable is defined in the source program, the type of the variable determines how much memory the compiler allocates. When the program executes, the variable consumes this amount of memory regardless of whether the program actually uses the memory allocated. This is particularly true for arrays. However, in many problems, it is not clear at the outset how much memory the program will actually need. Up to now, we have declared arrays to be "large enough" to hold the maximum number of elements we expect our application to handle. If too much memory is allocated and then not used, there is a waste of memory. If not enough memory is allocated, the program is not able to handle the input data.

We can make our program more flexible if, during execution, it could allocate additional memory when needed and free memory when not needed. Allocation of memory during execution is called *dynamic memory allocation*. C provides library functions to allocate and free memory dynamically during program execution. Dynamic memory is allocated on the heap by the system.

It is important to realize that dynamic memory allocation also has limits. If memory is repeatedly allocated, eventually the system will run out of memory.

14.2.1   Library Functions for Dynamic Allocation

Two standard library functions are available for dynamic allocation. The function `malloc()` allocates memory dynamically, and the function `free()` deallocates the memory previously allocated by `malloc()`. When allocating memory, `malloc()` returns a pointer which is just a byte address. As such, it does not point to an object of a specific type. A pointer type that does not point to a specific data type is said to point to `void` type, i.e. the pointer is of type `void *`. In order to use the memory to access a particular type of object, the void pointer must be cast to an appropriate pointer type. Here are the descriptions for `malloc()`, and `free()`:

malloc   *Prototype:* `void * malloc(unsigned size);`   *in:* `<stdlib.h and alloc.h>`

             Returns: void pointer to the allocated block of memory if successful, `NULL` otherwise

             Description: Returned pointer must be cast to an appropriate type.

free   *Prototype:* `void free(void * ptr);`   *in:* `<stdlib.h and alloc.h>`

             Returns: none

             Description: ptr must be a pointer to previously allocated block of memory

If successful, `malloc()` returns a pointer to the block of memory allocated. Otherwise, it returns a `NULL` pointer. One must always check to see if the pointer returned is `NULL`. If `malloc()` is successful, objects in dynamically allocated memory can be accessed indirectly by dereferencing the pointer, appropriately cast to the type of pointer required.

The size of the memory to be allocated must be specified, in bytes, as an argument to `malloc()`. Since the memory required for different objects is implementation dependent, the best way to specify the size is to use the `sizeof` operator. Recall that the `sizeof` operator returns the size, in bytes, of the operand.

For example, if the program requires memory allocation for an integer, then the size argument to `malloc()` would be `sizeof(int)`. However, in order for the pointer to access an integer object, the pointer returned by `malloc()` must be cast to an `int *`. The code takes the following form:

```
int *ptr;
ptr = (int *)malloc(sizeof(int));
```

Now, if the pointer returned by `malloc()` is not `NULL`, we can make use of it to access the memory indirectly. For example:

```
if (ptr != NULL)
    *ptr = 23;
```

Or, simply,

```
if (ptr)
    *ptr = 23;
printf("Value stored is %d\n", *ptr);
```

Later, memory allocated above may no longer be needed. In which case, it is important to free the memory. Thus:

```
free((void *) ptr);
```

deallocates the previously allocated block of memory pointed to by `ptr`. Or, more simply, we could write:

```
free(ptr);
```

`ptr` is first converted to `void *` in accordance with the function prototype, and then the block of memory pointed to by `ptr` is freed.

It is possible to allocate a block of memory for several elements of the same type by giving the appropriate value as an argument. Suppose, we wish to allocate memory for 100 float numbers. Then, if `fptr` is a `float *`, the following statement does the job:

```
fptr = (float *) malloc(100 * sizeof(float));
```

Pointer `fptr` points to the beginning of the memory block allocated, i.e. to the first object of the block of 100 float objects, `fptr + 1` points to the next float object, and so on. In other words, we have a pointer to an array of float type. The above approach can be used with data of any type including structures. The example in Figure 14.14 allocates memory for a structure, reads data into it, and then prints the data.

Sample Session:

```
***Dynamic Memory Allocation***

Student Name:  James J. Hillary
Student ID: 723
Student Name:  James J. Hillary ID: 723
```

```
/*    File: dynstruct.c
      This program uses dynamic allocation of a block of memory
      for an element of type stdrec structure. It then stores data
      for one student in the memory block, and prints out the data.
*/
#include <stdio.h>
#include <stdlib.h>

struct stdrec {
      char name[20];
      int id;
};

main()
{     struct stdrec * p;

      printf("***Dynamic Memory Allocation***\n\n");
      p = (struct stdrec *)malloc((sizeof(struct stdrec)));
      if (p) {
            printf("Student Name: ");
            gets(p->name);
            printf("Student ID: ");
            scanf("%d%*c", &p->id);
            printf("Student Name: %-10s  ", p->name);
            printf("ID: %4d\n", p->id);
      }
      else
            printf("Out of Memory\n");
}
```

Figure 14.14: Example program using a dynamic structure

14.2.2 Dynamic Arrays

Our next example allocates a block of memory dynamically for a number of elements of structure type. It reads data into the elements and prints the data. Once the returned pointer is cast to an appropriate type, the allocated memory block may be treated as an array of elements, with the returned pointer a pointer to the array. The code is shown in Figure 14.15.

Sample Session:

```
***Dynamic Arrays - Student Records***

Number of students:  2
Student Name:  James J. Hillary
Student ID: 723
Student Name:  John Paul Jones
Student ID: 321
^D
Student Name:  James J. Hillary ID: 723
Student Name:  John Paul Jones ID: 321
```

Dynamic memory allocation can also be performed by the library function `calloc()`, and the allocated memory freed as before by `free()`. All bytes in memory allocated by `calloc()` are cleared to zero, whereas memory allocated by `malloc()` is left unchanged. The description for `calloc()` is:

calloc   *Prototype:* `void * calloc(unsigned number, unsigned size);`   *in:* `<stdlib.h` `and alloc.h>`

Returns: void pointer to the allocated block of memory if successful, `NULL` otherwise

Description: Returned pointer must be cast to an appropriate type.

```
Example:
    void * ptr;          /* pointer to allocated block of memory */
    unsigned number;     /* number of elements to allocate */
    unsigned size;       /* size of memory to allocate in bytes */

    ptr = calloc(number, size);
```

We could have used `calloc()` in the previous program example as follows:

```
p = (struct stdrec *)calloc(n, sizeof(struct stdrec));
```

We could have then used the fact that the allocated memory is set to zero to signal the end of the number of elements in the effective array.

Normally, an array is defined with the range for each dimension specified, and memory is allocated at compile time. As we saw above, a single dimensional array of a desired size can be effectively defined at run time, i.e. during execution, using dynamic allocation. It is equally easy to define multi-dimensional arrays during execution by using dynamic allocation.

We first allocate an appropriate block of memory for the two dimensional array size desired. Since array storage in C is in row major form, we then treat the block as a sequence of rows with

```
/*    File: dynaray.c
      This program shows dynamic allocation of a block of memory
      for elements of the type struct stdrec. This is equivalent
      to allocating memory for an array of the specified size.
      The program reads in the number of students, allocates memory
      for that many structures, gets data for the students, and prints
      out the data.
*/
#include <stdio.h>
#include <stdlib.h>

struct stdrec {
      char name[20];
      int id;
};
void getdata(struct stdrec * p, int n);
void printdata(struct stdrec * p, int n);

main()
{     int n;
      struct stdrec * p;

      printf("***Dynamic Arrays - Student Records***\n\n");
      printf("Number of students: ");
      scanf("%d%*c", &n);
      p = (struct stdrec *)malloc(n * (sizeof(struct stdrec)));
      if (p) {
            getdata(p, n);
            printdata(p, n);
      }
      else
            printf("Out of Memory\n");
}
```

```
/* Gets data for n students */
void getdata(struct stdrec * p, int n)
{    int id, i;

     for (i = 0; i < n; i++) {
          printf("Student Name: ");
          gets(p->name);
          printf("Student ID: ");
          scanf("%d%*c", &p->id);
          p++;
     }
}

/* Prints data for all students */
void printdata(struct stdrec * p, int n)
{    int i;

     for (i = 0; i < n; i++) {
          printf("Student Name: %-10s  ", p->name);
          printf("ID: %4d\n", p->id);
          p++;
     }
}
```

Figure 14.15: Example code for a dynamic array of structures

```
/*     File: dyn2aray.c
       This program shows a dynamic specification of array size for
       a two dimensional array. Appropriate block of memory is then
       allocated. This block is then treated as a two dimensional
       array of the size specified.
*/

#include <stdio.h>
#include <stdlib.h>
void get2data(int * p, int rows, int cols);
void print2data(int * p, int rows, int cols);

main()
{     int cols, rows;
      int  *p;

      printf("***Dynamic Arrays - Two Dimensions***\n\n");
      printf("Type number of rows: ");
      scanf("%d", &rows);
      printf("Type number of columns: ");
      scanf("%d", &cols);
      p = (int *)malloc(rows * cols * sizeof(int));
      get2data(p, rows, cols);
      print2data(p, rows, cols);
}
```

the desired number of columns. The pointer to the allocated block is a pointer to the base type
of the array; therefore, it must be incremented to access the next column in a given row. It must
also be incremented to move from the last column of a row to the first column of the next row.

Figure 14.16 shows an example that asks the user to specify the number of rows and columns
for a two dimensional array. It then dynamically allocates a block of memory to accommodate the
array. The block is then treated as a two dimensional array with the specified rows and columns.
Data is read into the array, and then the array is printed. A sample output is shown below:

```
***Dynamic Arrays - Two Dimensions***

Type number of rows:   2
Type number of columns:   3
Type a row of integers with 3 columns:  1 2 3
Type a row of integers with 3 columns:  4 5 6
The array is:

              1         2         3
              4         5         6
```

```
/*   Gets data for a two dimensional array pointed to by int *, p,
     with specified rows and cols.
*/
void get2data(int * p, int rows, int cols)
{    int i, j;

     for (i = 0; i < rows; i++) {
          printf("Type a row of integers with %d columns: ", cols);
          for (j = 0; j < cols; j++) {
               scanf("%d", p);
               p++;
          }
     }
}

/*   Prints data in an array pointed to by int *, p, with
     specified rows and cols.
*/
void print2data(int * p, int rows, int cols)
{    int i, j;

     printf("The array is:\n");
     for (i = 0; i < rows; i++) {
          for (j = 0; j < cols; j++) {
               printf("%7d", *p);
               p++;
          }
          printf("\n");
     }
}
```

Figure 14.16: Dynamic allocation for 2D arrays

14.3   Pointers to Functions

We saw earlier that functions have a storage class and scope, similar to variables. In C, it is also possible to define and use function pointers, i.e. pointer variables which point to functions. Function pointers can be declared, assigned values and then used to access the functions they point to. Function pointers are declared as follows:

```
int (*fp)();
double (*fptr)();
```

Here, `fp` is declared as a pointer to a function that returns `int` type, and `fptr` is a pointer to a function that returns `double`. The interpretation is as follows for the first declaration: the dereferenced value of `fp`, i.e. `(*fp)` followed by `()` indicates a function, which returns integer type. The parentheses are essential in the declarations. The declaration without the parentheses:

```
int *fp();
```

declares a function `fp` that returns an integer pointer.

We can assign values to function pointer variables by making use of the fact that, in C, the name of a function, used in an expression by itself, is a pointer to that function. For example, if `isquare()` and `square()` are declared as follows:

```
int isquare(int n);
double square(double x);
```

the names of these functions, `isquare` and `square`, are pointers to those functions. We can assign them to pointer variables:

```
fp = isquare;
fptr = square;
```

The functions can now be accessed, i.e. called, by dereferencing the function pointers:

```
m = (*fp)(n);        /* calls isquare() with n as argument */
y = (*fptr)(x);      /* calls square() with x as argument */
```

Function pointers can be passed as parameters in function calls and can be returned as function values. Use of function pointers as parameters makes for flexible functions and programs. An example will illustrate the approach. Suppose we wish to sum integers in a specified range from x to y. We can easily implement a function to do so:

```
/*    File: sumutil.h */
int sum_int(int x, int y);

/*    File: sumutil.c */
#include <stdio.h>
#include "sumutil.h"
/*    Function sums integers from x to y. */
int sum_int(int x, int y)
{    int i, cumsum = 0;
```

```
        for (i = x; i <= y; i++)
              cumsum += i;
        return cumsum;
   }
```

The file `sumutil.h` contains prototypes for all the functions written in `sumutil.c`. Next, suppose we wish to sum squares of integers from x to y. We must write another function to do so:

```
   /*   File: sumutil.h - continued */
   int sum_squares(int x, int y);
   int isquare(int x);


   /*   File: sumutil.c - continued */
   /*   Function sums squares of integers form x to y. */
   int sum_squares(int x, int y)
   {    int i, cumsum = 0;

        for (i = x; i <= y; i++)
              cumsum += isquare(i);
        return cumsum;
   }


   /*   Function returns the square of x. */
   int isquare(int x)
   {
        return x * x;
   }
```

Function `isquare()` returns the integer square of `i`. The constructions of the two functions `sum_int()` and `sum_squares()` are identical. In both cases, we cumulatively add either the integers themselves or squares of the integers. A function `iself()`, which returns the value of the integer argument, can be used in `sum_int()` to make the functions truly identical. Here is a modified function that uses `iself()`:

```
   /*   File: sumutil.h - continued */
   int sum_integers(int x, int y);
   int iself(int x);


   /*   File: sumutil.c - continued */
   /*   Function sums integers from x to y. */
   int sum_integers(int x, int y)
   {    int i, cumsum = 0;

        for (i = x; i <= y; i++)
              cumsum += iself(i);
        return cumsum;
```

```
}

/*   Function returns the argument x. */
int iself(int x)
{
     return x;
}
```

The two sum functions, sum_integers() and sum_squares(), are now identical except for the functions used in the cumulative sum expressions. In one case, we use iself(), in the other case, isquare(). It is clear that a single more flexible *generic* sum function can be written by passing a function pointer, fp, as an argument with a value pointing to the appropriate function to use. The cumulative sum expression would then take the form:

```
cumsum += (*fp)(i);
```

Here is the implementation:

```
/*   File: sumutil.h - continued */
int sum_gen(int (*fp)(), int x, int y);

/*   File: sumutil.c - continued */
/*   Function sums values of *fp applied to integers from x to y. */
int sum_gen(int (*fp)(), int x, int y)
{    int i, cumsum = 0;

     for (i = x; i <= y; i++)
          cumsum += (*fp)(i);
     return cumsum;
}
```

Finally, we can improve the generic sum function by using a pointer to a function that updates the integer using a specified step size:

```
/*   File: sumutil.h - continued */
int sum(int (*fp)(), int x, int (*up)(), int step, int y);

/*   File: sumutil.c - continued */
/*   Function returns the sum of function *fp applied
     to integers from x to y, incremented by *up in step size.
*/
int sum(int (*fp)(), int x, int (*up)(), int step, int y)
{    int i, cumsum = 0;

     for (i = x; i <= y; i = (*up)(i, step))
          cumsum += (*fp)(i);
     return cumsum;
}
```

The function pointed to by (`*up`) takes two arguments, an integer to be updated and the step size. The generic function `sum()` can now be used to sum (`*fp`)(`i`) applied to integers `i`, which are updated by (`*up`)(`i, step`). The pointer variable, `fp` can point to any function that processes an integer and returns an integer. Similarly, `up` can point to any function that returns an updated integer value.

Let us now write a program that reads starting and ending integers as well as step size until EOF. For each set of data read, the program first computes and prints the sum of integers using `sum_int`, and sum of squares using `sum_squares()`. These two sums are in steps of one, since that is how the functions are written. Next, the program uses the above generic sum function `sum()` to compute sums of integers and squares in specified step sizes. Figure 14.17 shows the program. The update function used is `iincr()`, which merely returns `x` plus the step size. The program source files, `sums.c` and `sumutil.c`, are compiled separately and linked together. A sample run of the program is shown below:

Sample Session:

```
***Function Pointers - Sums of Integer Function Values***

Type starting, ending limits, and step size, EOF to quit
3 7 1
Sum of integers from 3 to 7 in steps of 1 = 25
Sum of squares from 3 to 7 in steps of 1 = 135
Sum of integers from 3 to 7 in steps of 1 is 25
Sum of squares from 3 to 7 in steps of 1 is 135
3 7 2
Sum of integers from 3 to 7 in steps of 1 = 25
Sum of squares from 3 to 7 in steps of 1 = 135
Sum of integers from 3 to 7 in steps of 2 is 15
Sum of squares from 3 to 7 in steps of 2 is 83
3 7 3
Sum of integers from 3 to 7 in steps of 1 = 25
Sum of squares from 3 to 7 in steps of 1 = 135
Sum of integers from 3 to 7 in steps of 3 is 9
Sum of squares from 3 to 7 in steps of 3 is 45
^D
```

For each set of input data, the output first shows sums of integers and squares in steps of one, and then in specified steps.

14.3.1 Function Pointers as Returned Values

It is also possible for functions to return a function pointer as a value. This ability increases the flexibility of programs. We will use a simple example to implement a function that returns a function pointer. The example is merely illustrative, and it would be easy to write a program to perform the same task without the use of a function pointer. Let us define a type, which is a pointer to a function that returns an integer.

```
typedef int (*PFI)();
```

```
/*    File: sums.c
      Other Source Files: sumutil.c
      Header Files: sumutil.h
      This program illustrates the use of function pointers to define a
      single function sum() that sums powers of integers between specified
      limits. The function is then applied to sum integers and squares.
      Individual functions to sum integers and squares are also implemented.
      The results are printed out for both approaches.
*/

#include <stdio.h>
#include "sumutil.h"

main()
{    int x, y, step, isquare(), iself(), iincr();

     printf("***Function Pointers - Sums of Function Values***\n\n");
     printf("Type starting, ending limits, and step size, EOF to quit\n");
     while (scanf("%d %d %d", &x, &y, &step) != EOF) {
          printf("Sum of integers from %d to %d in steps of 1 = %d\n",
                    x, y, sum_int(x, y));
          printf("Sum of squares from %d to %d in steps of 1 = %d\n",
                    x, y, sum_squares(x, y));
          printf("Sum of integers from %d to %d in steps of %d is %d\n",
                    x, y, step, sum(iself, x, iincr, step, y));
          printf("Sum of squares from %d to %d in steps of %d is %d\n",
                    x, y, step, sum(isquare, x, iincr, step, y));
     }
}

  /*   File: sumutil.h - continued */
  int iincr(int x, int step);

/*    File: sumutil.c - continued */
/*    Increments x by size of step. */
int iincr(int x, int step)
{
     return x + step;
}
```

Figure 14.17: Program illustrating function pointers

We can now use `PFI` as a data type in declaring variables, parameters, and returned values.

Our example program repeatedly reads an integer until `EOF`. If an integer is odd, the program computes its cube; otherwise, the program computes its square. For each integer, we call a function `evenodd()` which returns a function pointer either to `icube()` or to `isquare()` depending on whether the integer is odd or even. The function pointer returned by `evenodd()` and the integer itself are both passed to a function `process()`, which applies the dereferenced function pointer to the integer. The result is then printed. Figure 14.18 shows the program driver. For each integer, the program calls `evenodd()` to get a returned function pointer which is assigned to `fptr`. Then, it calls `process()` to apply `(*fptr)` to `x`. The result is then printed. Let us now write the function `evenodd()` that takes an integer as an argument. If the argument is odd, the function returns a pointer to `icube()`; otherwise, it returns a pointer to `isquare()`. The function `evenodd()`, together with functions `process()` and `icube()` are also shown in Figure 14.18.

When the program files `fptr.c` and `sumutil.c` are compiled and linked, the sample session is:

```
***Function Pointers - Squares and Cubes***

Type integers, EOF to quit
3
Integer = 3, power 2 or 3 = 27
5
Integer = 5, power 2 or 3 = 125
4
Integer = 4, power 2 or 3 = 16
^D
```

Using function pointers as parameters we can write generic functions. By returning function pointers, the called functions can select the functions that must be used in different circumstances. Function pointers help make a program compact as well as intelligent.

14.4 Summary

In this chapter we have discussed the concepts of storage class and scope for variables in a C program. The language provides four storage classes: automatic, register, external, and static. By default, variables declared in functions are of class `auto`, meaning that memory is allocated for them when the block is entered and automatically deallocated when the block is exited. Such variables may be referenced by name only within the block in which they are declared; i.e. they have *local* scope. Register storage class, declared with the class specifier, `register`, are a special case of automatic variables. This class suggests to the compiler that storage for the variable should be allocated in the CPU registers rather than memory. Use of this class should be limited to frequently referenced, time critical variables and only with familiarity with the particular architecture on which the program will be run.

External storage class is used for variables which should remain allocated for the entire execution of a program, and which have *global* scope. In using external variables, the operation of *defining* the variable (allocating memory for it) may be independent of *declaring* the variable (associating a name with the variable). An external variable must be defined exactly once, by

```
/*   File: fptr.c
     Other Source Files: sumutil.c
     Header Files: sumutil.h
     This program illustrates the use of function pointers, both as
     parameters in function calls and as returned values. Program
     reads integers until EOF. As each integer is read, the program
     calls a function evenodd() which returns a function pointer.
     This function pointer is then passed to process() to process
     the integer.

     evenodd() returns a pointer to isquare() if the argument is even,
     and to icube() otherwise. Function process() applies its first
     argument, which is a function pointer, to its second argument, which
     is an integer.
*/

#include <stdio.h>
#include "sumutil.h"
typedef int (*PFI)();
PFI evenodd(int x);
int process(PFI fp, int x);

main()
{    int x, y, z;
     PFI fptr;

     printf("***Function Pointers - Squares and Cubes***\n\n");
     printf("Type integers, EOF to quit\n");
     while (scanf("%d", &x) != EOF) {
          fptr = evenodd(x);
          y = process(fptr, x);
          printf("Integer = %d, power 2 or 3 = %d\n", x, y);
     }
}

/*   Function returns a function pointer. If x is odd, it returns
     a pointer to icube(). Otherwise, it returns a pointer to
     isquare().
*/
PFI evenodd(int x)
{    int isquare(), icube();

     if (x % 2)
          return icube;  /* icube is a pointer to function icube() */
     else
          return isquare; /* isquare is a pointer to isquare() */
}
```

```
/*    Function returns the result of applying the dereferenced function
      pointer fp to x.
*/
int process(PFI fp, int x)
{
      return (*fp)(x);    /* dereferenced fp applied to x */
}


/*    File: sumutil.h - continued */
int icube(int x);


/*    File: sumutil.c - continued */
/*    Function returns the cube of x. */
int icube(int x)
{
      return x * x * x;
}
```

Figure 14.18: Driver illustrating function pointer return values

specifying its type and name outside any function block. A declaration specified as **extern** declares the name of the variable without allocating storage, with the expectation that it has been defined elsewhere.

The storage class, *static*, is used for variables which have local scope, but which remain allocated for the entire program execution. Such variables, while local to a particular function, will retain their values across repeated calls and returns.

We have also seen how memory for variables of different storage classes is allocated in the memory of the computer: automatic variables are allocated and deallocated on the stack, whereas external and static variables are allocated from the heap.

In addition to storage allocated by the compiler, we have seen that additional storage can be allocated *dynamically* (i.e. at run time) using the **malloc()** or **calloc** system functions, and deallocated by the **free()** function. Data stored in dynamically allocated memory is always referenced indirectly.

Finally, we have expanded our discussion of functions; seeing that they have storage class, like variables; and that we can declare and access function indirectly through pointers. Functions are generally external and have global scope. However, we can limit the scope of a function to be within a single source file by declaring it to be of static storage class.

14.5   Problems

1. Modify the functions `getchr()` and `ungetchr()` of Section 14.1.6 so that any number of characters, up to a maximum of 40, can be put back into the input stream. Use these functions in a program that reads characters and puts all vowels back until a newline is read. At that point, the program writes the vowels that were put back in the input stream.

2. Write a program that reads scores from a file, but uses a dynamically allocated array. Assume that the first line of the file has the number of students. Read the value of the number of students, dynamically allocate an array for the scores, read the scores, and print them out.

3. Modify 2 so a student record is a structure. The file lists the number of students in the first line and the number of exams in the second line. Assume that an old weighted average is present as the last column in the file and that the first two columns are student name and an id number. Use dynamic allocation to write a menu-driven grading program that allows all possible options: add student, delete student, change grade, add new exam scores, compute various averages, etc.

4. Write a program that reads and sorts an array of numbers. Use a function to sort the array, but use a pointer to a function to make a comparison of two numbers. If the function returns True, swap the elements; otherwise, the elements are in correct order. Test the program with functions to sort in increasing and in decreasing order.

5. Write a program that reads an array of transliterated strings that represent equivalent strings in some language. The strings are to be sorted according to the alphabet of that language. First order the ASCII characters according to the alphabet of that language. Then use a function that returns True if two characters are ordered in a correct sequence. Use the function to sort the array of strings.

6. Use a structure with two members to represent either a complex number or a rational number. We will call each of them an ordered pair number. Write a generic function to add two ordered pair numbers where the addition is performed by a function a pointer to which is passed as an argument.

7. Repeat 6 to subtract two ordered pair numbers.

8. Repeat 6 to multiply two ordered pair numbers.

9. Repeat 6 to divide two ordered pair numbers.

10. Write a lexical analyzer that finds tokens of the following type in a string:

```
identifier
integer
float
operator
end of string
```

11. Repeat 10 without using an external variable. Use a pointer to a character as an argument to a function `get_token()` which indirectly returns the character read, but unused in a token.

12. Repeat 11, but use a static character variable in `get_token()` instead of indirectly returning the character read but unused. The static character will remain unchanged and may be used for the start of the next token.

Chapter 15

Engineering Programming Examples

In the preceding chapters we have presented the major features of the C language for declaring and accessing data, and controlling program execution flow including both the syntax required by the language and the semantics of the statements. We have also discussed "good" programming style and organization emphasizing the top down design process. In this chapter we make use of these features and techniques to develop several programs for commonly used operations in engineering and scientific computing.

We begin with with operations on matrices, including transforms and sums and products. We next discuss complex numbers together with their representation as a user defined data type and their uses. A program to find solutions to systems of linear algebraic equations is presented next using our complex number functions, followed by another common applications of complex number: the analysis of electrical circuits. We conclude the chapter with a program for numeric integration of arbitrary algebraic functions.

15.1 Matrices

We saw in Chapter 9 that systems of simultaneous linear algebraic equations can be represented and manipulated using two dimensional arrays. For example, a set of n equations in m unknowns:

$$a_{0,0} * x_0 + a_{0,1} * x_1 + \cdots + a_{0,m-1} * x_{m-1} = y_0$$
$$a_{1,0} * x_0 + a_{1,1} * x_1 + \cdots + a_{1,m-1} * x_{m-1} = y_1$$
$$\vdots$$
$$a_{n-1,0} * x_0 + a_{n-1,1} * x_1 + \cdots + a_{n-1,m-1} * x_{m-1} = y_{n-1}$$

Mathematically, such a system can be thought of in terms of a *matrix* equation written in the form:

$$A \times X = Y$$

where A is a matrix, i.e. a two dimensional array of coefficients $a_{i,j}$, X is a vector, i.e. a one dimensional array of elements x_j, and Y is also a vector, y_i. In a matrix representation of algebraic equations, the number of rows corresponds to the number of equations, and the number of columns corresponds to the number of unknowns. (In our case, the values of i range from 0 through $n-1$,

and those of j range from 0 through $m - 1$). When the number of rows and columns are equal, the matrix is square, otherwise the matrix is rectangular.

Such a matrix equation may be viewed as a *transformation* of a vector, X, to another vector, Y, by *matrix operator A*. Matrix formalism facilitates combinations of transformations, deriving properties of transformations, as well as finding solutions of equations.

In the next few sections, we will illustrate some useful matrix manipulations and begin to build our own library of utility functions for matrix operations. Many of the functions written can be used in a variety of programs; therefore, we will organize our code in several source files. The file `matutil.c` will contain all the functions we write for matrix manipulations. As usual, the prototypes for these functions are assumed to be in the file `matutil.h`.

In constructing our library, we first implement basic input/output functions for matrices and vectors: the function `readmatrix()` reads the elements of a matrix into a two dimensional array, and the function `printmatrix()` prints the matrix elements. (These functions are similar to the functions `getcoeffs()` and `pr2adbl()` in Chapter 9, except that the right hand side is not included in the matrix array). Vectors are read and printed by functions `readvector()` and `printvector()`. We assume the number of rows and columns are passed as parameters, and that the matrices are arrays of type `double`. The basic I/O functions for matrices and vectors and the requisite header files are straightforward to develop, and are shown in Figure 15.1 These functions are quite simple. The number of rows and columns for the two dimensional arrays are passed as parameters, as are the sizes of the one dimensional arrays. The functions `readmatrix()` and `readvector()` return a cumulative sum of the input values. If desired, these sums may be used by the calling function to detect a matrix or a vector with all zero elements.

15.1.1   Matrix Operations: Transforms

The first operation we will implement is the transformation of a vector X by a matrix A into a vector Y.

$$A \times X = Y$$

In other words, given the values of coefficients and the variables on the left hand side, find the values on the right hand side of the equations. Such linear transformation of a set of values is a common phenomenon in many practical applications such as electronic circuits, mechanical systems, chemical combinations, economic models, interactive relationships, and so forth.

If matrix A has r rows and c columns, the algorithm for the ith equation is:

```
y[i] = a[i][0]*x[0] + a[i][1]*x[1] +...+ a[i][c-1]*x[c-1]
```

This is applied for all the rows from 0 to $r - 1$. Translating this algorithm into C code, Figure 15.2 shows the function `mapvector()` that uses A to *map* (i.e. transform) vector X into vector Y.

With these utility functions in had, we can now write a driver program that reads a matrix and then transforms vectors until a zero vector is entered. The code is shown in Figure 15.3. The program declares all array ranges of size `MAX` and uses the function `getrc()` to read the number of rows and columns in the matrix. It then reads and prints the transform matrix of the specified size. Then, the program reads vectors until a zero vector is entered, and for each vector maps it

```
/* File: matdef.h */
#define MAX 10

/* File: matutil.h */
#include "matdef.h"
int readmatrix(double x[][MAX], int r, int c);
void printmatrix(double x[][MAX], int r, int c);
int readvector(double x[], int n);
void printvector(double x[], int n);

/*   File: matutil.c */
#include <stdio.h>
#include "matdef.h"
#include "matutil.h"
/*   Reads a matrix x with r rows and c columns. MAX
     provides the maximum column range for the array.
*/
int readmatrix(double x[][MAX], int r, int c)
{    int i, j;
     double z, sum = 0;

     printf("Matrix data entry:\n");
     for (i = 0; i < r; i++) {          /* for each row of matrix */
         printf("Type a row of %d numbers\n", c);

         for(j = 0; j < c; j++){        /* read c elements of the row */
             scanf("%lf", &z);
             x[i][j] = z;
             sum += z;
         }

     }
     return sum;
}

/*   Prints a matrix with r rows and c columns */
void printmatrix(double x[][MAX], int r, int c)
{    int i, j;

     printf("Matrix is:\n");
     for (i = 0; i < r; i++) {     /* for each row */

         for(j = 0; j < c; j++)   /* print the row */
             printf("%f  ",x[i][j]);

         printf("\n");
     }
}
```

```
/*   Reads a vector of size n. Function returns the sum
     of input values.
*/
int readvector(double x[], int n)
{    int i;
     double sum = 0;

     printf("Type %d numbers, <all zeros to quit>: ", n);
     for (i = 0; i < n; i++) {
          scanf("%lf", x + i);
          sum += x[i];
     }

     return sum;
}


/*   Prints a vector of size n. */
void printvector(double x[], int n)
{    int i;

     printf("Vector is:\n");

     for (i = 0; i < n; i++)
          printf("%f\n", x[i]);
}
```

Figure 15.1: Matrix and Vector I/O Functions

```
/*   File: matutil.h - continued */
void mapvector(double a[][MAX], double x[], double y[],
                int r, int c);



/*   File: matutil.c - continued */
/*   Computes a * x ====> y, where a[][] has r rows and c columns. */
void mapvector(double a[][MAX], double x[], double y[],
                int r, int c)
{    int i, j;

     for (i = 0; i < r; i++){
          y[i] = 0;

          for (j = 0; j < c; j++)
               y[i] += a[i][j] * x[j];
     }
}
```

Figure 15.2: Code for `mapvector()`

by `mapvector()` into a new vector which is printed. The function `getrc()` shown in Figure 15.4 and is included in file `matutil.c`. The source files `mat.c` and `matutil.c` are compiled and linked and tested producing the following sample session:

```
***Matrices and Vector Transformations***

Rows:   2
Columns:  3

Matrix data entry:
Type a row of 3 numbers
1 2 3
Type a row of 3 numbers
4 5 6

Matrix is:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000

Type 3 numbers, <all zeros to quit>:  1 2 3
Transformed Vector is:
14.000000
32.000000
```

```
/*   File: mat.c
     Other Source Files: matutil.c
     Header Files: matutil.h
     This program reads a matrix. It then repeatedly reads vectors.
     Each vector is transformed by the matrix and printed out.
*/

#include <stdio.h>
#include "matdef.h"
#include "matutil.h"

main()
{    double a[MAX][MAX];
     double x[MAX], y[MAX];
     int r, c;

     printf("***Matrices and Vector Transformations***\n\n");
     getrc(&r, &c);
     readmatrix(a, r, c);
     printmatrix(a, r, c);

     while (readvector(x, c)) {
         mapvector(a, x, y, r, c);
         printf("Transformed "); /* Prefix to printvector() mesg */
         printvector(y, r);
     }
}
```

Figure 15.3: Driver to read and transform vectors

```
/*   File: matutil.h - continued */
void getrc(int * rp, int * cp);



/*   File: matutil.c - continued */
/*   Gets the number of rows and columns for a matrix. rp point to
     rows and cp points to columns.
*/
void getrc(int * rp, int * cp)
{
     printf("Rows: ");
     scanf("%d", rp);
     printf("Columns: ");
     scanf("%d", cp);
}
```

Figure 15.4: Code for `getrc()`

```
Type 3 numbers, <all zeros to quit>:   3 2 1
Transformed Vector is:
10.000000
28.000000

Type 3 numbers, <all zeros to quit>:   2.5 3 4.5
Transformed Vector is:
22.000000
52.000000

Type 3 numbers, <all zeros to quit>:   0 0 0
```

15.1.2   Matrix Operations: Sums and Products

Other common manipulations involving matrices require addition of two matrices, multiplication of two matrices, and inversion of matrices. In this section, we will implement matrix addition and matrix multiplication algorithms. Addition of two matrices may arise when two sets of equations relate the same set of variables. For example, consider the matrix equations:

$$A \times X = Y1$$
$$B \times X = Y2$$

Corresponding equations of the two sets can be added together to obtain a combined single set:

$$C \times X = Y$$

```
/*   File: matutil.h - continued */
void matsum(double c[][MAX], double a[][MAX],
            double b[][MAX], int rows, int cols);



/*   File: matutil.c - continued */
/*   Adds matrix a to matrix b to generate a matrix c. Parameters
     r and c specify the rows and columns.
*/
void matsum(double c[][MAX], double a[][MAX],
            double b[][MAX], int rows, int cols)
{    int i, j;

     for (i = 0; i < rows; i++)
          for (j = 0; j < cols; j++)
               c[i][j] = a[i][j] + b[i][j];
}
```

Figure 15.5: Code to add rectangular matrices

where, in matrix terms,

$$C \;=\; A + B$$
$$and$$
$$Y \;=\; Y1 + Y2$$

Vectors are special cases of rectangular matrices having n rows and 1 column. We will therefore implement a single function that sums two rectangular matrices. The sum of matrices A and B generates a new matrix, say C. If the elements of matrix A are a[i][j], and those of B are b[i][j], then the sum matrix, C with elements c[i][j], is determined as follows:

    c[i][j] = a[i][j] + b[i][j]

The implementation of matrix addition is easy, and the code is shown in Figure 15.5.

Multiplication of two matrices A and B results when combining two transformations, i.e. where a vector being transformed by a matrix A is itself the result of a transformation by a matrix B. Consider the following two sets of equations:

$$A \times Z \;=\; Y$$
$$and$$
$$B \times X \;=\; Z$$

Since, by the second equation, Z equals $B \times X$, we can substitute $B \times X$ for Z in the first equation:

$$A \times B \times X = Y$$

Or, the combined equation results in:

$$C \times X = Y$$

The product of matrices A and B generates a matrix C. If the number of rows and columns of A are given by $r1$ and $c1$, and those of B are given by $r2$ and $c2$, then, the number of elements of Z represents the number of columns of A and the number of rows of B, i.e. $c1 = r2$. Also, C must have the same number of rows as A and the same number of columns as B. That is, the number of rows and columns of C must be $r1$ and $c2$. It turns out that each `c[i][j]` is a result of a scalar product of row i of matrix A and column j of matrix B. Let the ith row of A and the jth column of B be:

```
a[i][0]  a[i][1]  ...  a[i][c1 - 1]
b[0][j]  b[1][j]  ...  b[r2 - 1][j]
```

then, the scalar product, `c[i][j]`, is given by:

```
a[i][0] * b[0][j] + a[i][1] * b[1][j] + ... + a[i][c1-1] * b[r2-1][j]
```

With this algorithm, the sum is easily implemented as a cumulative sum, initialized to zero. Each pass through the loop adds one product term, `a[i][k] * b[k][j]`, for k from zero through $c1-1$. That is, the following loop computes `c[i][j]`:

```
c[i][j] = 0;
for (k = 0; k < c1; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Such a loop is repeated for all appropriate i rows and j columns. The code for matrix product is shown in Figure 15.6.

We can now write a simple example that uses the matrix functions defined above as shown in Figure 15.7. The program adds and multiplies matrices. To keep the program simple, we assume square matrices. The program first reads in the size of square matrices, and then reads in the two matrices. These matrices are added and multiplied, and the resultant matrices are printed. A sample session is shown below:

```
***Square Matrices - Sums and Products***

Size of square matrices:  2

Matrix data entry:
Type a row of 2 numbers
2 3
Type a row of 2 numbers
3 4
```

```
/*   File: matutil.h - continued */
void matprod(double c[][MAX], double a[][MAX], double b[][MAX],
             int r1, int c1, int r2, int c2);


/*   File: matutil.c - continued */
/*   Matrix multiplication of matrix a (r1 rows and c1 columns)
     and matrix b (r2 rows and c2 columns). Result is matrix c with
     r1 rows and c2 columns.
*/

void matprod(double c[][MAX], double a[][MAX], double b[][MAX],
             int r1, int c1, int r2, int c2)
{    int i, j, k;

     if (c1 != r2) {
         printf("Error - Columns of matrix A do not match rows of B\n");
         return;
     }

     for (i = 0; i < r1; i++)
         for (j = 0; j < c2; j++) {
             c[i][j] = 0;

             for (k = 0; k < c1; k++)
                 c[i][j] += a[i][k] * b[k][j];
         }
}
```

Figure 15.6: Code for matrix product

```
/*    File: matops.c
      Other Source Files: matutil.c
      Header Files: matutil.h
      This  program adds and multiplies two square matrices.
      The matrices are read into two dimensional arrays.
*/
#include <stdio.h>
#include "matdef.h"
#include "matutil.h"
main()
{     double a[MAX][MAX], b[MAX][MAX], c[MAX][MAX];
      int n;

      printf("***Square Matrices - Sums and Products***\n\n");
      printf("Size of square matrices: ");
      scanf("%d", &n);

      readmatrix(a, n, n);
      readmatrix(b, n, n);
      matsum(c, a, b, n, n);

      printf("Sum ");            /* Prefix to msg in printmatrix() */
      printmatrix(c, n, n);
      matprod(c, a, b, n, n, n, n);

      printf("Product ");      /* Prefix to msg in printmatrix() */
      printmatrix(c, n, n);
}
```

Figure 15.7: Driver to test matrix operations

```
Matrix data entry:
Type a row of 2 numbers
4 5
Type a row of 2 numbers
6 7

Sum Matrix is:
6.000000 8.000000
9.000000 11.000000

Product Matrix is:
26.000000 31.000000
36.000000 43.000000
```

Another important matrix operation is the inversion of a square matrix. An inverse matrix has the property:

$$A^{-1} \times A = A^{-1} \times A = I$$

where A^{-1} is the *inverse matrix* and I is a *unit matrix* with unit diagonal elements and zero elements elsewhere. The unit matrix has the property:

$$I \times X = X \times I = X$$

If $A \times X = Y$, it follows that

$$A^{-1} \times A \times X \;=\; A^{-1} \times Y$$
$$or$$
$$X \;=\; A^{-1} \times Y$$

Thus, given the inverse matrix, the solution to the matrix equation for any Y is easily obtained.

Inversion of a matrix is somewhat more complex. An inverse of a matrix can be obtained by the Gauss-Jordan method — a modified version of the Gauss elimination method discussed in Chapter 9. A good reference [1], for matrix computational methods as well as other numeric methods, is given at the end of this chapter.

15.2   Complex Numbers

Complex numbers are encountered in many mathematical applications. In this section, we will first review complex numbers and operations involving complex numbers. We will then represent complex numbers using structure types and implement many of the basic complex number operations.

The squares of a real number, either positive or negative, is a positive number. Numbers whose squares are negative cannot be real numbers; they are, therefore, called *imaginary numbers*. Thus,
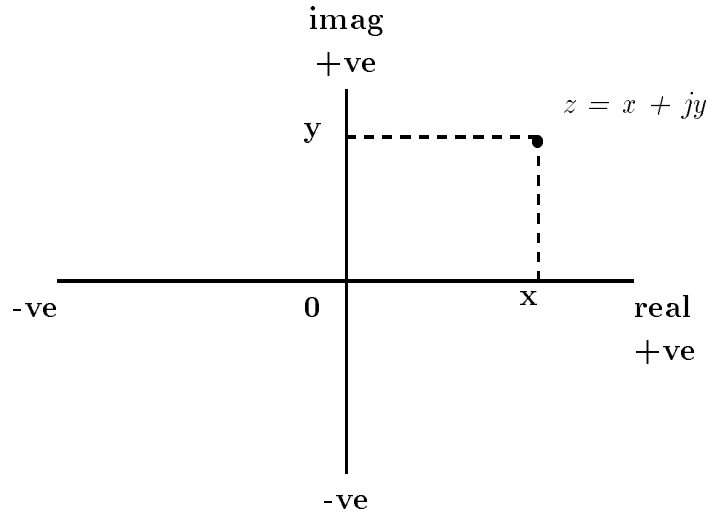
Figure 15.8: Complex Numbers in Rectangular Coordinates

imaginary numbers are the square roots of negative numbers. For example, consider:

$$z = \sqrt{-|x|}$$

Here $|x|$ is the absolute value of x, and thus z is a square root of a negative number, i.e. an imaginary number. Imaginary numbers are written in a normalized manner as follows:

$$\begin{aligned} z &= \sqrt{-1 \cdot |x|} \\ &= \sqrt{-1} \cdot \sqrt{|x|} \\ &= j \cdot y \end{aligned}$$

where, $j = \sqrt{-1}$, and $y = \sqrt{|x|}$. Square root of -1 is represented by the special symbol, i in mathematics or j in Electrical Engineering. Thus, an imaginary number is represented by j times a real number y. A *complex number* is a number that is sums of both a real and an imaginary number.

$$z = x + j \cdot y$$

Both x and y are real numbers, and z is a complex number. Either of the real numbers x or y can, of course, be zero; in which case, the complex number reduces to either a real or an imaginary number. The number x is called the *real part* of z, and y is called the *imaginary part*. Remember that both the real part, x, and the imaginary part, y, are real numbers. It is j that is an imaginary number, not y.

Complex numbers can be visualized geometrically as points on a two dimensional plane with rectangular axes, **real** and **imag**. Then, the real part of a number is the projection of the point onto the real axis, and the imaginary part of the number is the projection onto the imaginary axis imag (see Figure 15.8. For these reasons, the complex number representation as a sum of real and imaginary parts is called *representation in rectangular coordinates*.

Addition, subtraction, multiplication, and division operators for complex numbers are defined in terms of the same operator symbols as for real numbers, viz. **+**, **-**, *****, **/**. The sum of two

complex numbers is simply the sum of their real parts plus j times the sum of their imaginary parts. Thus, if

$$
\begin{aligned}
z1 &= x1 + j \cdot y1 \\
z2 &= x2 + j \cdot y2
\end{aligned}
$$

then the sum of `z1` and `z2` is given by:

$$
z1 + z2 = (x1 + x2) + j \cdot (y1 + y2)
$$

The product of `z1` and `z2` is obtained by multiplying the two numbers, replacing $j \cdot j$ by -1, and collecting the real terms and the imaginary terms. Thus,

$$
\begin{aligned}
z1 * z2 &= (x1 + j \cdot y1) \cdot (x2 + j \cdot y2) \\
&= (x1 \cdot x2 - y1 \cdot y2) + j \cdot (x1 \cdot y2 + x2 \cdot y1).
\end{aligned}
$$

Division of two complex numbers is a little more involved. First, we define the *complex conjugate*, z^*, of a number, $z = x + j \cdot y$, as one with the same real part, x, but whose imaginary part is $-y$. Thus, the complex conjugate of z is:

$$
z^* = x - j \cdot y
$$

Observe that the product of z and z^* is real:

$$
\begin{aligned}
z \cdot z^* &= (x \cdot x + y \cdot y) + j \cdot (x \cdot y - x \cdot y) \\
&= x \cdot x + y \cdot y.
\end{aligned}
$$

Now, we can divide two complex numbers:

$$
\frac{z1}{z2} = \frac{x1 + j \cdot y1}{x2 + j \cdot y2}
$$

To separate the result into real and imaginary parts, we first make the denominator real by multiplying both the numerator and the denominator by $z2^*$.

$$
\begin{aligned}
\frac{z1}{z2} &= \frac{z1 \cdot z2^*}{z2 \cdot z2^*} \\
&= \frac{(x1 + j \cdot y1) \cdot (x2 - j \cdot y2)}{x2 \cdot x2 + y2 \cdot y2} \\
&= \frac{x1 \cdot x2 + y1 \cdot y2}{x2 \cdot x2 + y2 \cdot y2} + j \cdot * \frac{-x1 \cdot y2 \cdot x2 \cdot y1}{x2 \cdot x2 + y2 \cdot y2}
\end{aligned}
$$

With this description of complex numbers and operations on them, we would like to develop programs that can work with them. Complex number is not a native data type in C, but we would like to represent complex numbers in a program as if it were. We will define an *abstract data type*, `complex`, using `typedef` and define functions to serve as operators on complex numbers.

We will represent complex numbers as ordered pairs of real and imaginary parts, (using rectangular form defined above), and implement the ordered pairs as structures. We will use `typedef` to define a data type, `rect`, for this structure. (We choose the name `rect` because `complex` data type with an identical structure is already defined in `math.h`. We can, of course, use the `complex` type defined in `math.h`, but we define a `rect` type to illustrate the use of `typedef`). Figure 15.9 shows this definition and the functions for addition and multiplication of complex numbers. We use type `double` in the structure `rect` for greater precision in computation. In a similar manner, it is easy to write the remaining functions for subtraction and division of two complex numbers. Implementation of these functions is left as an exercise.

```
/*    File: compdef.h */
struct rect {
     double real;
     double imag;
};

typedef struct rect rect;

/*    File: computil.h */
rect addc(rect z1, rect z2);
rect multc(rect z1, rect z2);

/*    File: computil.c */
#include <stdio.h>
#include <math.h>    /* math function protos: sqrt(), atan(), etc. */
#include "compdef.h"
#include "computil.h"

/*    Returns a sum of two complex numbers - rect form. */
rect addc(rect z1, rect z2)
{    rect z;

     z.real = z1.real + z2.real;
     z.imag = z1.imag + z2.imag;
     return z;
}

/* Returns a product of two complex numbers - rect form. */
rect multc(rect z1, rect z2)
{    rect z;

     z.real = z1.real * z2.real - z1.imag * z2.imag;
     z.imag = z1.real * z2.imag + z1.imag * z2.real;
     return z;
}
```

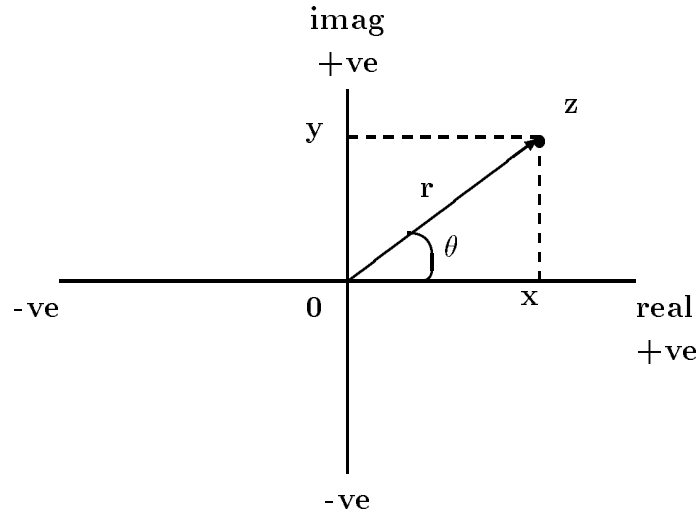Figure 15.9: Complex number utility functions

Figure 15.10: Complex Numbers in Polar Coordinates

15.2.1   Complex Numbers and Vectors

It is also possible to represent a point on a two dimensional plane in terms of polar coordinates. Polar coordinates are given in terms of:

1. the length, r, of the (directional) line from the origin to the point, and

2. the counterclockwise angle, θ, that the line makes with the reference axis, namely the positive horizontal axis.

The directional line of length r at an angle θ with respect to the reference axis is called a vector (See Figure 15.10). The projection of the vector onto the real axis is $r \cdot \cos(\theta)$, and the projection onto the imaginary axis is $r \cdot \sin(\theta)$. Thus, a complex number, represented by the pair (r, θ) in polar coordinates, can be written in rectangular coordinates as:

$$
\begin{aligned}
z & = r \cdot \cos(\theta) + j \cdot r \cdot \sin(\theta) \\
  & = x + j \cdot y
\end{aligned}
$$

Thus, the real and imaginary parts, x and y, in terms of r and θ are:

$$
\begin{aligned}
x & = r \cdot \cos(\theta) \\
y & = r \cdot \sin(\theta)
\end{aligned}
$$

Since,

$$
\exp(j \cdot \theta) = \cos(\theta) + j \cdot \sin(\theta)
$$

z can also be written as:

$$
z = r \cdot \exp(j \cdot \theta)
$$

As we shall soon see, this exponential form is convenient for multiplication and division.

Given rectangular coordinates x and y, we can determine r and θ as follows. We know:

$$
\begin{aligned}
x^2 + y^2 &= r^2 \\
\frac{y}{x} &= \tan(\theta)
\end{aligned}
$$

so,

$$
\begin{aligned}
r &= \sqrt{x^2 + y^2} \\
\theta &= \arctan(\frac{y}{x})
\end{aligned}
$$

Observe that the length, r, is the square root of $z \cdot z^*$. The length r is called the *magnitude* of the vector, and the angle θ is called the *angle* or *phase angle* of the vector.

As we have seen, addition and subtraction of complex numbers is easy to perform in rectangular coordinates. On the other hand, multiplication and division of two complex numbers in rectangular coordinates is not so easy. Conversely, it is easy to perform multiplication and division in polar coordinates. Given that two numbers are:

$$
\begin{aligned}
p1 &= r1 * \exp(j \cdot \theta 1) \\
p2 &= r2 * \exp(j \cdot \theta 2)
\end{aligned}
$$

It is easy to see that:

$$
\begin{aligned}
p1 \cdot p2 &= r1 \cdot r2 \cdot \exp(j \cdot (\theta 1 + \theta 2)) \\
\frac{p1}{p2} &= \frac{r1}{r2} \cdot \exp(j \cdot (\theta 1 - \theta 2)).
\end{aligned}
$$

From this analysis, we can implement complex numbers in polar coordinates as shown in Figure 15.11 together with functions for multiplication and division in polar coordinates. It is also important to be able to convert back and forth between rectangular and polar coordinates. It is easy to write the necessary conversion routines to convert complex numbers in rectangular coordinates to polar coordinates, and vice versa — they are shown in Figure 15.12. The function `polar_to_rect()` is quite straight forward; `rect_to_polar()` uses the arc tangent function `atan()` defined in the standard library. This function returns an angle in the range $-\pi/2$ to $\pi/2$, thus we need to adjust the angle when the real part is zero and when it is negative. If the real part is zero, the angle is $\pi/2$ if the imaginary part is positive, and $-\pi/2$ if it is negative. Next, if the real part is negative, the angle must be incremented by π. Since we use many standard library trigonometric functions, the file `math.h` must be included at the head of `computil.c` and we must link the math library when the program is compiled.

These functions provide a useful library for processing with complex numbers. Let us now make use of them in two application programs.

15.2.2 Roots of Algebraic Equations

One such application where complex numbers occur is in finding roots of algebraic equations. A *linear* algebraic equation of the form:

$$a * x + b = 0$$

```
/* File:  compdef.h - continued */
struct polar {
    double r;
    double theta;
};

typedef struct polar polar;



/*   File: computil.h - continued */
polar multp(polar p1, polar p2);
polar divp(polar p1, polar p2);

/*   File:  computil.c - continued */
/*   Returns a product of complex numbers - polar form. */
polar multp(polar p1, polar p2)
{    polar p;

    p.r = p1.r * p2.r;
    p.theta = p1.theta + p2.theta;
    return p;
}

/* Returns p1 / p2 - polar form. */
polar divp(polar p1, polar p2)
{    polar p;

    p.r = p1.r / p2.r;
    p.theta = p1.theta - p2.theta;
    return p;
}
```

Figure 15.11: Complex number utility functions in polar coordinates

```
/*   File: computil.h - continued */
rect polar_to_rect(polar p);
polar rect_to_polar(rect z);


/* File: computil.c - continued */
/* Returns the rect form of a number in polar form. */
rect polar_to_rect(polar p)
{    rect z;

     z.real = p.r * cos(p.theta);
     z.imag = p.r * sin(p.theta);
     return z;
}


/* Returns the polar form of a number in rect form. */
#define PI 3.14159
polar rect_to_polar(rect z)
{    polar p;

     p.r = sqrt(z.real * z.real + z.imag * z.imag);
     if (z.real == 0)
          p.theta = z.imag >= 0 ? PI / 2 : - PI / 2;
     else
          p.theta = atan(z.imag / z.real);
     if (z.real < 0)
          p.theta = PI + p.theta;
     return p;
}
```

Figure 15.12: Conversion from polar to rect and rect to polar

in one unknown variable, x, can be easy to solve depending on the values of the coefficients, a and b. If $a = 0$ and $b = 0$, the equation is homogeneous and has no unique solution; any value for x will make the equation true. If $a = 0$ but b is non-zero, the equation has no solution; no value of x will make it true. Otherwise, if a is non-zero, the solution for x is easily determined:

$$x = -b/a$$

A *quadratic* equation is a polynomial of second degree in x of the form:

$$a \cdot x^2 + b \cdot x + c = 0$$

If a is zero, the equation reduces to a linear equation that is easy to solve. If a is non-zero, there are two solutions:

$$
\begin{aligned}
x1 &= \frac{-b + \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a} \\
x2 &= \frac{-b - \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}
\end{aligned}
$$

The form of the solutions depends on the *discriminant*:

$$b^2 - 4 \cdot a \cdot c$$

If the discriminant is positive, the square root is a real number and the roots, $x1$ and $x2$ are both real numbers. If the discriminant is zero, the two roots are real and equal. Otherwise, if the discriminant is negative, the square root is an imaginary number and the roots are complex numbers:

$$
\begin{aligned}
x1 &= \frac{-b}{2 \cdot a} + j \cdot \frac{\sqrt{4 \cdot a \cdot c - b^2)}}{2 \cdot a} \\
x2 &= \frac{-b}{2 \cdot a} - j \cdot \frac{\sqrt{4 \cdot a \cdot c - b^2)}}{2 \cdot a}
\end{aligned}
$$

In fact, the two roots are complex conjugates: the real parts are the same, the imaginary parts are negatives of each other. Complex roots of polynomials with real coefficients always occur in complex conjugate pairs.

We will now implement a program that finds the roots of a quadratic equation, and then tests each root by evaluating the quadratic polynomial for that value of the variable. If the value is a root, the polynomial must evaluate to zero. When testing roots, we must be able to evaluate the polynomial for all possible values of roots, including complex values. For consistency in testing, we will represent all roots as complex numbers with real roots having a zero imaginary part. Therefore, we will need a function to force a real number into a complex number, as well as a function to make a complex number given its real and imaginary parts. These functions are shown in Figure 15.13, and are added to the file `computil.c` with their prototypes in `computils.h`. Finally, since complex numbers are not a native data type in C, we will also need a function to print complex numbers in the accepted form. If the number is real, it must print only the real part. If the number is imaginary, it must print only j times the imaginary part. Otherwise, it must print a complex number as $a + j \cdot b$ or $a - j \cdot b$, depending on the sign of the imaginary part. The function is also shown in Figure 15.13.

```
/*   File: computil.h - continued */
rect make_rect(double x, double y);
rect force_rect(double x);
void print_rect(rect z);

/*   File: computil.c - continued */
/*   Makes a complex number in rect form. */
rect make_rect(double x, double y)
{    rect z;

     z.real = x;
     z.imag = y;
     return z;
}

/*   Forces a real number to a complex number - rect form. */
rect force_rect(double x)
{    rect z;

     z.real = x;
     z.imag = 0;
     return z;
}

/*   Prints a complex number in rect form. */
void print_rect(rect z)
{
     if (z.real == 0 && z.imag == 0)    /* if number is zero */
         printf("0");                    /* print zero. */
     if (z.real != 0)                /* print real part, if non-zero */
         printf("%f ", z.real);
     if (z.imag != 0) {             /* print imag part, if non-zero */
         if (z.imag > 0)
             printf("+ j * %f", z.imag);
         else if (z.imag < 0)
             printf("- j * %f", -z.imag);
     }
     printf("\n");
}
```

Figure 15.13: Code for make_rect() and force_rect()

With all of these utility functions completed, the program logic is now simple to implement. It reads in the coefficients a, b, c of the quadratic equation and uses the function findroots() to find the roots of the quadratic. The function forces the roots to complex form and returns them indirectly. The arguments of findroots() are the coefficients of the quadratic, and pointers to the two roots. The program then uses the function eval_quad() to verify each root by evaluating the quadratic polynomial at that value. The arguments of eval_quad() are the coefficients of the quadratic, and the value at which the quadratic is to be evaluated. The code for the driver is shown in Figure 15.14. For each set of coefficients, main() checks if a is zero and b is non-zero, in which case it prints that the equation is linear with root -c/b. Otherwise, if both a and b are zero, it prints an invalid equation message, and in either case continues to read the next set of coefficients. On the other hand, if a is non-zero, the driver calls findroots() to find the roots as complex numbers and returns them by indirectly to z1 and z2. Each root is printed and verified using eval_quad(). The process continues until end of file.

We next implement the function findroots() shown in Figure 15.15. It computes the roots, forces them to complex numbers and returns the values through the pointer parameters.

Finally, we write eval_quad() to evaluate a quadratic polynomial at a given complex value of the unknown variable. Since the value of the unknown, x is complex, we force all coefficients to complex numbers before using our utility functions addc() and multc(). To reduce the number of multiplications required to evaluate the polynomial we perform the expression

$$a \cdot x^2 + b \cdot x + c = a \cdot x \cdot c + b \cdot x + c$$
$$= (((a \cdot z) + b) \cdot z) + c$$

The function is shown in Figure 15.16 The complex variable, w, is initialized to zero and then used for the cumulative complex sum of the polynomial. As we saw in Chapter 5, due to errors in rounding and floating point number representation, our result may not be precisely zero. Therefore, eval_quad() checks that w.real and w.imag are sufficiently close to zero using the library function fabs() to verify the solution and print an appropriate message. A sample run of the program is shown below:

```
***Roots of Quadratic Equations***

Quadratic Equation:  a * x * x + b * x + c = 0
Type coefficients a b c, EOF to quit
2 3 5
z1 = -0.750000 + j * 1.391941
The value is verified as a root of the equation
z2 = -0.750000 - j * 1.391941
The value is verified as a root of the equation
1 2 1
z1 = -1.000000
The value is verified as a root of the equation
z2 = -1.000000
The value is verified as a root of the equation
2 2 5
z1 = -0.500000 + j * 1.500000
The value is verified as a root of the equation
```

```
/*    File; roots.c
      Other Source Files: computil.c
      Header Files: compdef.h, computil.h
      This program finds the roots of quadratic equations. For each
      equation, the program verifies that the roots make the
      quadratic polynomial evaluate to zero. All roots, including real
      roots, are treated as complex roots.
*/

#include <stdio.h>
#include <math.h>          /* needed in this file and in computil.c */
#include "compdef.h"       /* defines rect and polar types */
#include "computil.h"      /* prototypes for functions in computil.c */

void eval_quad(double a, double b, double c, rect z);
void findroots(double a, double b, double c, rect *zp1, rect *zp2);

main()
{     rect z1, z2;
      double a, b, c, x;

      printf("***Roots of Quadratic Equations***\n\n");
      printf("Quadratic Equation: a * x * x + b * x + c = 0\n");
      printf("Type coefficients a b c, EOF to quit\n");
      while (scanf("%lf %lf %lf", &a, &b, &c) != EOF) {
           if (a == 0) {
               if (b != 0) {
                   printf("Linear equation - root is %f\n", - c / b);
                   continue;
               }
               else {
                   printf("Invalid equation\n");
                   continue;
               }
           }
           else
               findroots(a, b, c, &z1, &z2);
           printf("z1 = ");
           print_rect(z1);
           eval_quad(a, b, c, z1);
           printf("z2 = ");
           print_rect(z2);
           eval_quad(a, b, c, z2);
      }
}
```

Figure 15.14: Code for quadratic solver driver

```
/*   File; roots.c - continued */
/*   Finds the roots of a quadratic equation. Roots are forced
     to complex values and stored where zp1 and zp2 point.
*/
void findroots(double a, double b, double c, rect *zp1, rect *zp2)
{    double discr, x, x1r, x2r, x1i, x2i;
     rect z1, z2;

     x = 2 * a;
     discr = b * b - 4 * a * c;
     if (discr >= 0) {
         x1r = -b / x + sqrt(discr) / x;
         x2r = -b / x - sqrt(discr) / x;
         x1i = x2i = 0;
     }
     else {
         x1r = x2r = -b / x;
         x1i = sqrt(-discr) / x;
         x2i = -x1i;
     }
     z1 = make_rect(x1r, x1i);
     z2 = make_rect(x2r, x2i);
     *zp1 = z1;
     *zp2 = z2;
}
```

Figure 15.15: Code for findroots()

```
/*    File; roots.c - continued */
/*    Function evaluates a quadratic equation with x equal to
      the unknown variable.
*/
void eval_quad(double a, double b, double c, rect x)
{    rect w = {0, 0};

     w = multc(force_rect(a), x);  /* a * x */
     w = addc(w, force_rect(b));   /* a * x + b */
     w = multc(w, x);              /* a * x * x + b * x */
     w = addc(w, force_rect(c));   /* a * x * x + b * x + c */
     if (fabs(w.real) < 0.000001 && fabs(w.imag) < 0.000001)
          printf("The value is verified as a root of the equation\n");
     else
          printf("The value is not a root of the equation\n");
}
```

Figure 15.16: Code for `eval_quad()`

```
z2 = -0.500000 - j * 1.500000
The value is verified as a root of the equation
^D
```

15.2.3 Impedance of Electrical Circuits

Another important application of complex numbers is in computing impedances of electrical circuits. The basic components of such circuits are resistors, inductors, and capacitors as shown in Figure 15.17. These devices can be connected in series or parallel to make more complex circuits as shown in Figure 15.18 where each component has an impedance, Z. In general, the impedance is modeled as a complex quantity depending on their value and the value of the angular frequency, ω, in radians per second, of the electrical signal for which the impedance is to be computed. The impedance of a resistor of R ohms is simply R, that of an inductor of L henrys is $j \cdot \omega \cdot L$, and that of a capacitor of C farads is $\frac{-j}{(\omega \cdot C)}$.

The impedance of a series or a parallel combination of sub-circuits is defined in terms of the individual impedances of the sub-circuits. The impedance of a series combination of impedances, $Z1$ and $Z2$ is the sum of the individual impedances, i.e. $Z1 + Z2$. The impedance of a parallel combination of impedances $Z1$ and $Z2$ is the reciprocal of the reciprocal sum of the individual impedances:

$$\frac{1}{\frac{1}{Z1} + \frac{1}{Z2}}$$

Let us first write a set of circuit utility functions to determine:

- the impedance of a basic component,
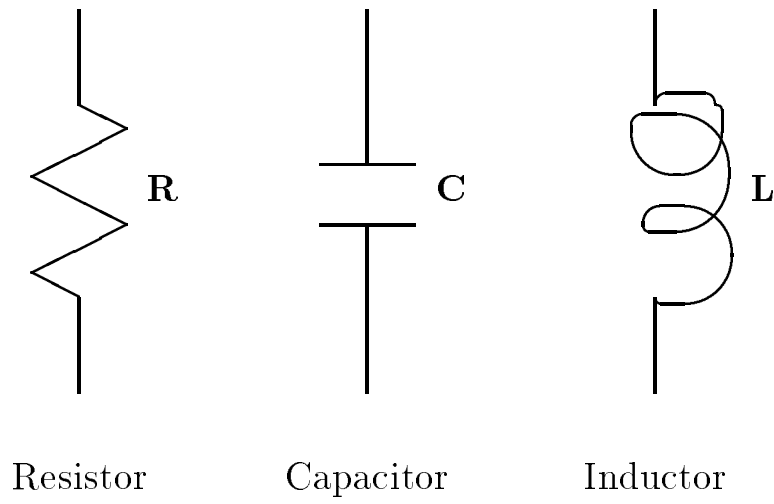
- the impedance of a series combination,

Resistor   Capacitor   Inductor

Figure 15.17: Basic Electrical Circuit Components
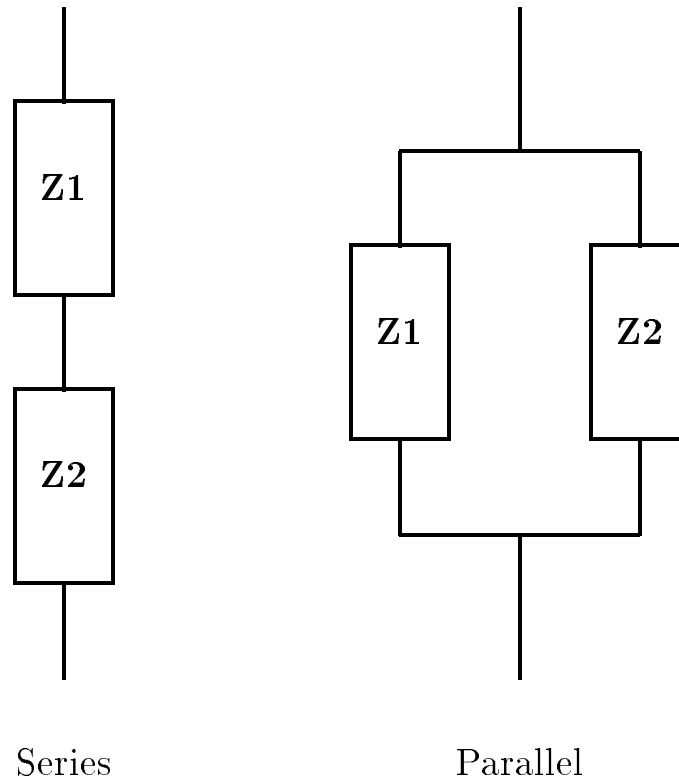

Series   Parallel

Figure 15.18: Series and Parallel Combinations

- and the impedance of a parallel combination.

We will use the complex data type, rect, as defined in compdef.h as well as the functions defined in computil.c.

The function comp_imped() determines the impedance of a basic component whose element type (a character) and value are passed together with the value of the angular frequency ω (we will call w). The code is shown in Figure 15.19. The only point to note here is that if w * C is zero, the impedance is infinite. It is not possible to handle an infinite value in computers, so some garbage value is returned. The calling program must handle a zero value of w * C as a special case. Next, we implement the functions that compute the series and parallel combination of impedances shown in Figure 15.20. The function series() merely returns the sum of the two impedances. The function parallel() uses polar coordinates to compute the reciprocals of impedances. It is much easier to compute the reciprocal of a complex number in polar coordinates than in rectangular coordinates; whereas complex numbers are easier to sum in rectangular coordinates. Conversion routines are used to convert polar to rectangular, and vice versa.

*

We are now ready to implement a program to compute the impedance of an electrical circuit. Let us assume a circuit which is a series combination of two sub-circuits as shown in Figure 15.21. The first sub-circuit is a series combination of resistor **R1** and inductor **L**. The second sub-circuit is a parallel combination of resistor **R2** and capacitor **C**. Figure 15.22 shows the program to find the impedances of this circuit for different sets of values of **R1**, **R2**, **L**, **C**, and ω. The program reads a set of values for **R1**, **R2**, **L**, **C**, and ω. It calls series() to compute the impedance z1 of **R1** and **L** in series. If ωC is zero, the impedance of the capacitor is infinite; so the impedance of the parallel combination, z2, is just the impedance of **R2**. Otherwise, parallel() is called to compute the impedance z2 of **R2** and **C** in parallel. In all cases, comp_imped() is used to compute the impedances of the basic components and series() is called to compute the impedance of z1 and z2 in series. The values of these impedances are printed by print_rect(). A sample run is shown below:

```
***Impedance of Electrical Circuits***

Ckt:  a series combination of:
R1 and L in series, and
R2 and C in parallel.
Type values of R1 R2 L C W, EOF to quit
1 1 1 1 1
Impedance of series branch z1 = 1.000000 + j * 1.000000
Impedance of parallel branch z2 = 0.500000 - j * 0.499999
Overall impedance z = 1.500000 + j * 0.500001
10 10000 0.01 0.000001 10000
Impedance of series branch z1 = 10.000000 + j * 100.000000
Impedance of parallel branch z2 = 1.000023 - j * 99.990005
Overall impedance z = 11.000023 + j * 0.009995
^D
```

The second circuit values represent a circuit near resonance. Its impedance is almost purely resistive, since the imaginary part is close to zero.

```
/*   File: cktutil.h */
rect comp_imped(int component, double value, double w);

/*   File: cktutil.c */
#include <stdio.h>
#include <math.h>
#include "compdef.h"
#include "computil.h"
#include "cktutil.h"
/*   Returns the impedance for each of the components R, L, C. */
rect comp_imped(char component, double value, double w)
{    rect z;
     double x;

     switch(component) {
          case 'r': z = force_rect(value);   /* impedance is R */
                    break;
          case 'l': z.real = 0;
                    z.imag = w * value;      /* impedance is j * w * L */
                    break;
          case 'c': z.real = 0;
                    x = w * value;           /* x = w * C */
                    /* if x is non-zero, impedance is -j/(w*C) */
                    if (x)
                         z.imag = - 1 / x;
                    else ;            /* else, impedance is infinite */
                    break;            /* handle separately */
     }
     return z;
}
```

Figure 15.19: Code for `comp_imped()`

```
/*   File: cktutil.h - continued */
rect series(rect z1, rect z2);
rect parallel(rect z1, rect z2);


/*   File: cktutil.c - continued */
/*   Returns the impedance of a series combination of impedances
     z1 and z2: sum of z1 and z2.
*/
rect series(rect z1, rect z2)
{
     return addc(z1, z2);
}


/*   Returns the impedance of a parallel combination of impedances
     z1 and z2: reciprocal of the sum of 1 / z1 and 1 / z2.
*/
rect parallel(rect z1, rect z2)
{    polar p1, p2, p;
     rect z;

     p1 = rect_to_polar(z1);
     p1.r = 1 / p1.r;                    /* reciprocal of z1 */
     p1.theta = -p1.theta;
     p2 = rect_to_polar(z2);
     p2.r = 1 / p2.r;                    /* reciprocal of z2 */
     p2.theta = -p2.theta;
     z = addc(polar_to_rect(p1), polar_to_rect(p2));   /* sum reciprocals
*/
     p = rect_to_polar(z);
     p.r = 1 / p.r;                      /* take reciprocal of the sum */
     p.theta = -p.theta;
     z = polar_to_rect(p);          /* convert to rect. */
     return z;                      /* return in rect form */
}
```
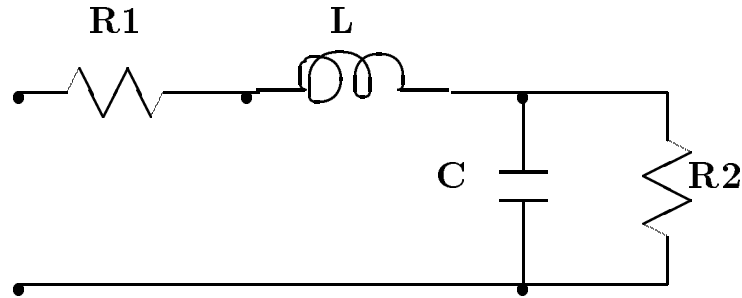
Figure 15.20: Code for series() and parallel()

Figure 15.21: An Example Circuit

15.3 Integrals

Another common operation that arises in engineering and scientific computing is integration. While software application packages exist to perform symbolic integration, some functions do not lend themselves to such a "closed form" method. A common computing method for approximating the value of an integral in *numeric integration*. In this section we will develop a small program implementing Simpson's Rule for numeric integration.

The integral of a function between specified limits gives the area under the function curve as shown in Figure 15.23. Numeric methods can approximate the area under the curve by summing approximate sub-areas under linearized parts of the function at uniformly sampled points (Figure 15.24). The smaller the sampling interval, h, the greater the precision of the computed integral. An algorithm to evaluate such an integral may be written in terms of the value of the function at sample points between the two limits. For example, assume the limits of integration for function, $f(x)$ are $x = a$ and $x = b$. Then, the function values between the two limits at intervals of h are:

$$f(a), f(a + h), f(a + 2h), \ldots, f(b)$$

The total number of samples is

$$\frac{(b - a)}{h}$$

There are many methods to approximate the value of an integral in terms of these sample values. Simpson's Rule gives a fairly accurate integral of function $f(x)$ between specified limits a and b:

$$Integralvalue = \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + y_n)$$

where, $y_k = f(a + kh)$ for $k = 0, 1, 2, \ldots, n$, h is the sampling interval, and $n = \frac{b-a}{h}$ with h adjusted so that n is an even integer. Except for the multiplier $\frac{h}{3}$, the above sum is called the *Simpson sum*. Observe that in the Simpson sum, sample values of the function evaluated at odd k, i.e. $y_1, y_3, y_5, \ldots$, are multiplied by 4, and sample values at even values of k, except for y_0 and y_n, are multiplied by 2. Finally, sample values y_0 and y_n are added without a multiplier.

We will now slightly modify the concept of a generic sum from Chapter 14 to implement a function that numerically evaluates an integral of a specified function between two limits, i.e. modify the generic function, `sum()`, into a generic Simpson sum function. Since integral computation requires real numbers, we use type `double` for all our computation. The parameters to `simpsum`

```
/*    File; imped.c
      Other Source Files: computil.c, cktutil.c
      Header Files: compdef.h, computil.h, cktutil.h
      This program finds the impedance of an electrical circuit for
      different values of the components and the frequency. The
      circuit consists of a series of two sub-circuits: a series
      combination of a resistor R1 and an inductor L, and a parallel
      combination of a resistor R2 and a capacitor C. The values of
      these components are specified by the user together with the
      angular frequency w in radians per second. The impedance is found
      for each user specified set of values until EOF.
*/
#include <stdio.h>
#include <math.h>
#include "compdef.h"
#include "computil.h"
#include "cktutil.h"
main()
{     rect z, z1, z2;
      double r1, r2, l, c, w;

      printf("***Impedance of Electrical Circuits***\n\n");
      printf("Ckt: A series combination of:\n");
      printf("          R1 and L in series, and\n");
      printf("          R2 and C in parallel.\n");
      printf("Type values of R1 R2 L C w, EOF to quit\n");

      while (scanf("%lf %lf %lf %lf %lf",
                   &r1, &r2, &l, &c, &w) != EOF) {
          z1 = series(comp_imped('r', r1, w), comp_imped('l', l, w));
          if (w == 0 || c == 0)
               z2 = comp_imped('r', r2, w);
          else
               z2 = parallel(comp_imped('r', r2, w),
                             comp_imped('c', c, w));
          z = series(z1, z2);
          printf("Impedance of series branch z1 = ");
          print_rect(z1);
          printf("Impedance of parallel branch z2 = ");
          print_rect(z2);
          printf("Overall impedance z = ");
          print_rect(z);
      }
}
```

Figure 15.22: Driver program for an example circuit
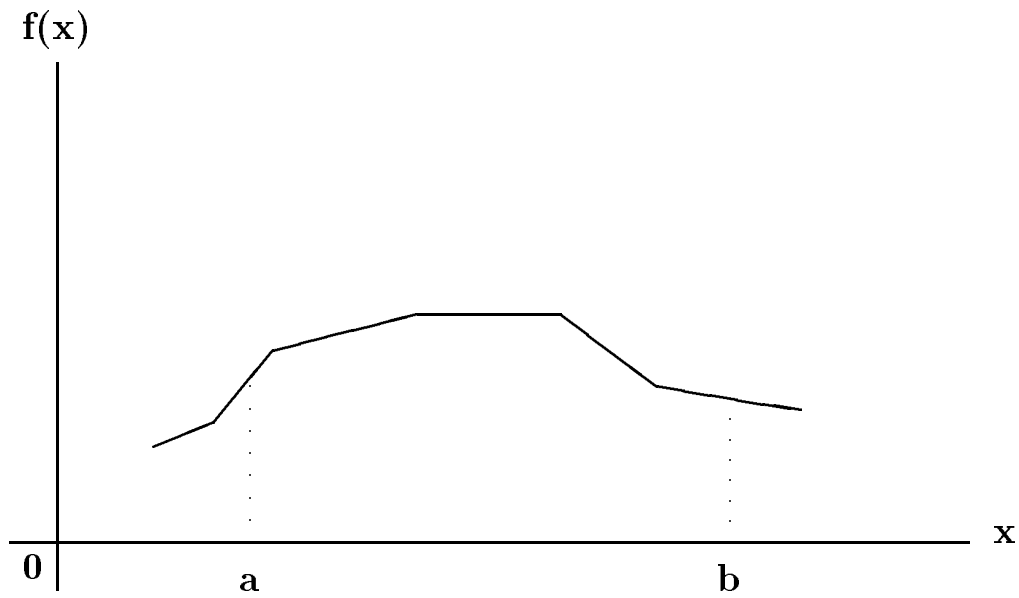
Figure 15.23: Integral of a Function from a to b
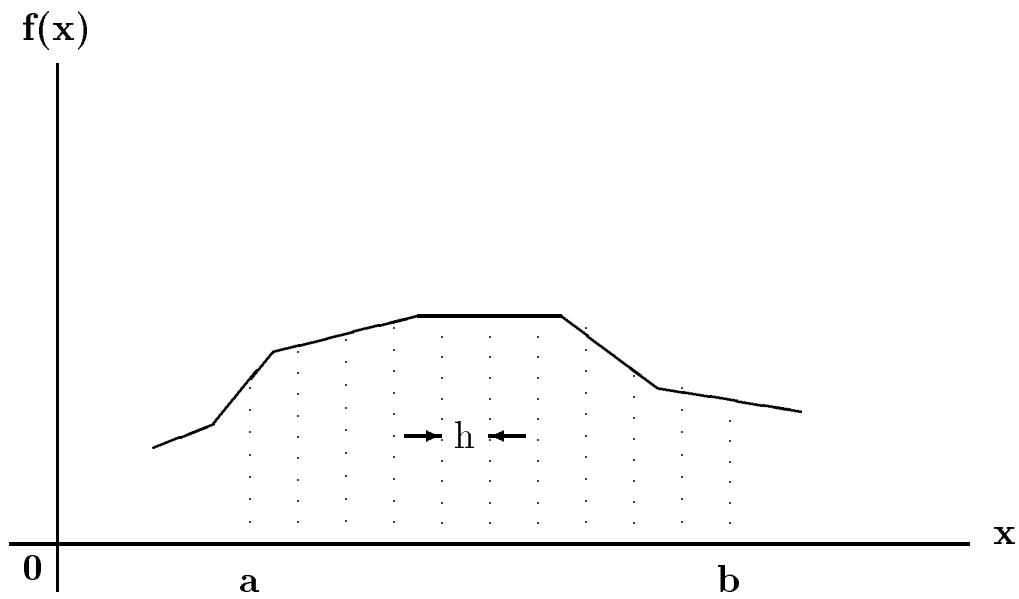
Figure 15.24: Function Sampling at Intervals of h

```
/*   File: simputil.h */
double simpsum(double (*fp)(), double a, double (*up)(),
               double step, double b);


/*   File: simputil.c */
#include <stdio.h>
#include "simputil.h"
/*   Returns the Simpson sum of *fp from a to b. */
double simpsum(double (*fp)(), double a, double (*up)(),
               double step, double b)
{    double i, cumsum = 0;
     int m;

     for (i = a, m = 0; i < b; m++, i = (*up)(i, step)) {
          if (m == 0)
               cumsum += (*fp)(i);
          else if (m % 2)
               cumsum += 4 * (*fp)(i);
          else
               cumsum += 2 * (*fp)(i);
     }
     cumsum += (*fp)(b);
     return cumsum;
}
```

Figure 15.25: Code to compute the Simpson sum

are the function pointer, fp, a lower limit, a, an update function pointer, up, a sampling interval, step, and an upper limit, b. The code is shown in Figure 15.25.

The integer variable, m represents the sample number. If m is zero, the function sample is added to the cumulative sum, if it is odd, the sample value times 4 is added to the cumulative sum; otherwise, sample value times 2 is added. Finally, the sample value y_n at b is added and the resulting Simpson sum is returned. It is easy now to implement the function integral() to compute the integral of a function between limits a and b. It merely gets the Simpson sum, multiplies by step/3 and returns it as seen in Figure 15.26 The update function incr() merely returns the value of its first argument increased by the value of the second argument, step. This function is included in sumutil.c together with other useful functions, self(), square(), and cube() shown in Figure 15.27.

Finally, we write a simple driver that computes integrals of several functions using integral() shown in Figure 15.28. The program first reads the sampling interval, h; then repeatedly reads the integration limits until EOF. For each set of limits, it calculates the number of samples, n, for the specified h. Since the Simpson sum requires an even number of samples, n is increased by one if it is odd, and the sampling interval h is adjusted to correspond to the even value of n. Then, the program computes the integral by calling integral() for three different functions: a straight

```
/*    File: simputil.h - continued */
double integral(double (*fp)(), double a, double b, double step);

/*    File: simputil.c - continued */
/*    Computes integral of a function *fp from a to b in sample
      steps of step.
*/
double integral(double (*fp)(), double a, double b, double step)
{    double r, incr();

     r = simpsum(fp, a, incr, step, b);
     return r * step / 3;
}
```

Figure 15.26: Code for `integral()`

line $f(x) = x$, a square, $f(x) = x^2$, and a cube, $f(x) = x^3$. The values of integrals are printed.
The program is in three source files, which must be compiled and linked: `integr.c`, `sumutil.c`,
and `simputil.c`. Here are two sample sessions with different sampling intervals:

```
***Integration by Simpson's Rule***

Integrals of x, square of x, and cube of x
Sampling interval for integration:   0.1
Type lower and upper limits, EOF to quit
0 1
Integral of st.  line = 0.566667
Integral of square = 0.400000
Integral of cubic = 0.316667
^D

***Integration by Simpson's Rule***

Integrals of x, square of x, and cube of x
Sampling interval for integration:   0.01
Type lower and upper limits, EOF to quit
0 1
Integral of st.  line = 0.500000
Integral of square = 0.333333
Integral of cubic = 0.250000
^D
```

Remember, the smaller the sampling interval, the greater the accuracy of the computed integral.
The first session specifies a fairly large sampling interval of 0.1 and the results are not very accurate.
The exact answers for the integrals are 0.5, 0.3333, and 0.25. The second session specifies a

```
/*   File: sumutil.h - continued */
double self(double x);
double square(double x);
double cube(double x);
double incr(double x, double step);

/*   File: sumutil.c - continued */
/*   Returns x. */
double self(double x)
{
    return x;
}

/*   Returns square of x. */
double square(double x)
{
    return x * x;
}

/*   Returns cube of x. */
double cube(double x)
{
    return x * x * x;
}

/*   Returns x incremented by step. */
double incr(double x, double step)
{
    return x + step;
}
```

Figure 15.27: Code for self(), cube(), and incr()

```
/*   File: integr.c
     Other Source Files: sumutil.c, simputil.c
     Header Files: sumutil.h, simputil.h
     This program computes definite integrals of several functions
     between specified limits. Parameters of integral() are: a
     function pointer, limits, and number of samples. It returns
     the integral of that function. Integrals of straight line,
     square, and a cubic are printed out for specified limits.
     Simpson's Rule is used to compute integral of a function f(x)
     between limits a and b as follows:

         I = (h / 3) * (y0 + 4y1 + 2y2 + 4y3 + 2y4 +...+ yn),

     where, yk = f(a + kh), and h = (b - a) / n for some even integer
     n. Except for the multiplier h/3, the above sum is called the
     Simpson sum.
*/

#include <stdio.h>
#include "sumutil.h"
#include "simputil.h"

main()
{    double  r, a, b, h, self(), square(), cube();
     int n;

     printf("***Integration by Simpson's Rule***\n\n");
     printf("Integrals of x, square of x, and cube of x\n");
     printf("Sampling interval for integration: ");
     scanf("%lf", &h);
     printf("Type lower and upper limits, EOF to quit\n");
     while (scanf("%lf %lf", &a, &b) != EOF) {
          n = (b - a) / h;
          if (n % 2) {
               n++;
               h = (b - a) / n;
          }
          r = integral(self, a, b, h);
          printf("Integral of st. line = %f\n", r);
          r = integral(square, a, b, h);
          printf("Integral of square = %f\n", r);
          r = integral(cube, a, b, h);
          printf("Integral of cubic = %f\n", r);
     }
}
```

Figure 15.28: Driver for Numeric Integration Program

somewhat better sampling interval 0.01, and the results are quite accurate. A smaller sampling interval would be even better, but would require more computation time. A compromise between accuracy and speed is required in most numeric computations.

15.4 Summary

In this chapter we have shown how we can use the features of the C language as well as the program design techniques we have discussed throughout this text to implement programs for several common engineering and scientific applications. In general, we have done this by developing a set of utility functions to use as a toolbag for writing the application. Our treatment of engineering and scientific computing has not been, by any means, comprehensive. References, such as [1] below, can be a source of algorithms for many additional applications. However, with your current knowledge of C you can now develop programs from these algorithms to solve **your** problems.

<div align="center">

E hoʻomaikaʻi ʻoukou.
(Good luck).

</div>

References:

[1]. Press, William H., Flannery, Brian P., Teukolsky, Saul A., Vettering, William T., Numerical Recipes in C, Cambridge University Press, Cambridge, 1988.

15.5   Problems

1. Write a menu driven program that allows the user to specify a matrix operation: add, subtract, multiply.

2. Write a simple calculator program that performs complex number arithmetic. The input should be an operand, followed by an operator, followed by an operand. The output should be the result of applying the operator to operands.

3. Repeat 2, but allow the user to continue entering operators and operands in sequence. The user may also request that a value should be saved for later use.

4. Evaluate a polynomial $P(z)$ with specified coefficients for a complex value of the variable. The highest degree of the polynomial is 10. The user must enter coefficient and exponent pairs for the polynomial, and specify the value of the variable for which the polynomial is to be evaluated.

5. Consider a rational function of a variable s:

$$\frac{P(s)}{Q(s)}$$

where $P(s)$ and $Q(s)$ are polynomials in a variable, s, with real coefficients. Evaluate the function for a value of $s = j\omega$. Evaluate the function for different values of s. Plot the magnitude and angle of the values.