The History of C

You are about to be introduced to a powerful programming language that is known cryptically as C. This language is attracting considerable attention worldwide because the software industry is adopting the language to great advantage. One reason for this popularity is portability. That is to say, a program written in C can be transferred easily from one computer to another with minimal changes or none at all. Programs written in C are fast and efficient. This versatility makes C a desirable language in the highly competitive software industry.

The C language was developed at Bell Laboratories in the early 1970's by a systems programmer named Dennis Ritchie. It was written originally for programming under an operating system[1] called UNIX, which itself was later rewritten almost entirely in C.

In case you have wondered why a language should be given such a cryptic name as C, it derives from the fact that it is based on an earlier version written by Ken Thompson, another Bell Laboratories systems engineer. He adapted it from a language that was known by the initials BCPL, which stand for Basic Combined Programming Language. To distinguish his version of the language from BCPL, Thompson dubbed it B, the first of the initials BCPL. When the language was modified and improved to its present state, the second letter of BCPL, C, was chosen to represent the new version. It is mere coincidence that the letters B and C are in alphabetic order. Some contend that the successor to C will be D, while others say it should be called P. You are left to ponder this weighty problem for yourself.

There is no single compiler for C. Many different organizations have written and implemented C compilers, which differ from one another to a greater or lesser extent. Even for the same computer, there may be several different compilers, each with its own particular requirements. As of this writing, C is in the process of being officially standardized. Even so, you should be prepared for minor variations that may become apparent. Compilers can be very sensitive even to minor differences of syntax. If you experience any difficulties, just consult the manual that comes with your version of C. Or better still, get help from someone with first-hand experience.

The C language is often described as a "middle-level" language. It permits programs to be written in much the same style as that of most modern high-level languages, such as FORTRAN, COBOL, BASIC, PL/I, and Pascal. Where it differs is that C permits very close interaction with the inner workings of the computer. It is anologous to a car that has the luxury of automatic gears, but at the option of the

[1] An operating system is a set of programs that enable the user to interact with the hardware components of a computer system.

driver, permits the manual shifting of gears. It is possible in C to deal with the machine at a fairly low level. Nevertheless, C is a general-purpose structured programming language that has much in common with the best of the high-level languages. Even though C is concise and terse (as its short name seems to suggest), it is also extremely powerful.

When you learn how to write programs in C, those programs will also have to be compiled and executed. For details on how to edit, compile, and run programs, consult the manuals for your particular compiler and computer. Whenever you feel you are ready, just start typing in your programs. You may proceed at your own speed. The chances are rather high that you will make mistakes at first, but this is par for the course. The more errors you make at the beginning, the fewer you will make later on, so don't get too depressed. Just rest assured that there is nothing that you can possibly do within a program to cause any damage to the computer, even if the results leave something to be desired.

The Structure of a Simple Program

Program 1-2 is a very simple C program. Before we say anything at all about it, examine it for yourself and try to discover what it does.

```
main ()
{
    printf("Welcome to C programming!\n");
}
```

Program 1–2

Now here is the output that this program produced. It prints out the string of characters enclosed by the double quotation marks (but not the quotes themselves). How close did you come?

```
Welcome to C programming!
```

Output 1–2

If you are familiar with other programming languages, this program may look strange to you. One reason is that almost every character in the program is lowercase. In most other languages, uppercase letters are the rule. So if you come from a background of another high-level language, such as Pascal, you will have to become accustomed to lowercase. You should also remember that C (unlike Pascal) "knows" the difference between uppercase and lowercase letters. Lowercase letters are characteristic of the C language.

C program instructions, like those of Pascal, are written in free format. That is to say, they may appear anywhere on a line. Spacing generally is of no consequence in C, nor are there many restrictions on where to end a line. Program 1-2 could just as easily have been written as follows:

```
main(){printf("Welcome to C programming!\n");}
```

Needless to say, when written in this "scrunched-up" fashion, the program is not very easy to read. But, so far as the C compiler is concerned, the second version is perfectly valid and would produce the same output. Now let us examine the program very carefully.

The first line of Program 1-2 specifies a function (defined for now as a group of program statements) named *main*. This is a special name that is recognized by the system under which C runs: It points to the precise place in the program where execution begins. Every C program must have a *main* function.

The word *main* is followed by a pair of ordinary parentheses. Do not try to use square brackets ([ ]) or even curly braces ({ }). Although these other symbols are used in C, they are not used in this context. In the example, there is nothing enclosed within the parentheses. Later on we shall have much more to say about the significance of any information that may be contained within these parentheses.

Next comes the { symbol, called the left brace or left curly bracket. It matches the right brace, which is located all the way at the end of the *main* function. (It may be interpreted as meaning, "Brace yourself for the function which follows immediately!") The braces enclose the body of the function (that is to say, the statements that form the function). In this case, the body consists of only one statement, the one beginning with *printf*. In a short while, however, we shall be including many statements between the two braces.

The *printf* Function

The *printf* function is used to print out a message, either on screen or paper. (The letter "f" in *printf* could stand for either "formatted" or "function.") It is equivalent to the WRITE statement in Pascal, only more powerful.

The statement between the braces in Program 1-2 specifies that the function called *printf* is to be called into action. In C jargon, we say that the function is *called* (or *invoked*). In this example, the function has a single *literal*, or *character string* as its argument (the value upon which it operates). This string is the sentence contained within the parentheses following the word *printf*. The contents of this string, called the *control string*, are printed out.

Notice that a character string in C is a series of characters written within double quotation marks. These punctuation marks are not actually part of the literal, but serve to delimit it. For this reason, double quotation marks cannot be used directly within a literal, because their presence would confuse the compiler (if not the programmer).

If you need to include double quotes as part of a literal, you must precede it with a backward slash, also called a backslash. This symbol tells the compiler that the quotation marks are part of the string, as in the following example:

```
"He said, \"Hi!\""
```

Notice that the string ends with two double quotation marks. The first is part of the literal, whereas the second serves to delimit it. The backslash itself does not become part of the literal, but merely tells the compiler: "The following mark is *not* a delimiter." (Incidentally, when reading program listings, be sure not to confuse the double quote, which is a single character on the keyboard, with two adjacent apostrophes.)

Please note also that a blank space in a string counts as a character. After all, it is inserted in a program by pressing the space bar.

Unlike Pascal, C restricts the programmer to a single control string in the *printf* statement. That is to say, the statement

```
printf("Oh, ","say ","can ","you ","C?","\n");
```

would *not* work, even though it may look reasonable enough. All the strings except the first would be ignored.

The *newline* Character

In Program 1-2, the control string ends with the symbols \n. These two adjacent symbols together represent the *newline* character. (We will henceforth refer to them as the *newline* character, although strictly speaking, the character and its representation are not the same thing.) Even though the *newline* character is composed of two separate symbols, they are translated by the compiler into a single character.

The *newline* is analogous to a typewriter's carriage return, which advances to the beginning of the next line. The *newline* symbol instructs the computer to advance to the next new line before printing subsequent information. The *printf* function does not do this automatically. Note that the *newline* character is enclosed within the control string.

The *newline* character is an example of an escape sequence in C. We will have much more to say about escape sequences later on. Note the direction of the backslash symbol. C also uses another symbol called the slash (or forward slash). It is very easy to confuse these two symbols, so try to get them straight at the outset.

What do you think would be the result of having a succession of \n characters? You're absolutely right; it would cause blank lines to appear in the output.

The Use of the Semicolon

The end of the *printf* line is denoted by a semicolon. In C, *all* statements are terminated with a semicolon. Novices to C frequently forget to add the semicolon, so be on your guard!

The Use of Braces

The end of the program is marked off by the closing brace, which matches the opening brace we mentioned previously. Notice that these braces line up with the letter *m* of the word *main.* This alignment is not strictly necessary, but it makes the program more readable. A stranger to the program can see quickly what the program does by casting a glance at the contents of the program enclosed within these two braces. Positioning the braces in this way has now become conventional. Notice, too, that the body of the function is indented within the braces. Once again, this makes no difference to the computer or the compiler, but it helps us to read the program more easily.

We shall now expand our horizons and examine a C program containing two *printf* statements (Program 1-3). The first one does not contain the \n symbol. How do you think this will affect the way the output is printed?

```
main()
{
    printf("Give me land, lots of land");
    printf("And the starry skies above...\n");
}
```

Program 1–3

```
Give me land, lots of landAnd the starry skies above...
```

Output 1–3

It is clear that the appearance of the output leaves much to be desired. The output consists of a single line with the last word of the first *printf* statement (*land*) printed alongside the first word (*And*) of the second *printf.* This happened because the control string does not end with the \n symbol in the first *printf* statement. In its absence, the computer assumes that the next piece of information should be printed immediately after the first. To avoid this problem, all that is necessary is to rewrite the first line as follows:

```
printf("Give me land, lots of land\n");
```

As soon as the word *land* is printed, the *newline* character sends the printer to the beginning of the next new line.

An alternative is to place the *newline* character at the beginning of the literal in the second *printf* statement. If this is done, the carriage return is effected before the second literal is printed.

The Use of Comments in a Program

Program 1-4 prints a poem. It also demonstrates the method used in C to add a comment or a helpful remark to a program. This is done by beginning the comment with the two characters /* and ending it with the characters */. No space can be included between the two characters. Between these pairs of characters (delimiters), any characters may be included in either uppercase or lowercase. In other words, a comment appears as follows:

```
/* Whatever you want using any characters at all */
```

The only restriction is that a comment may not contain the closing delimiter */ as part of the comment's text.

All such comments are ignored by the compiler and have no effect on the way the program runs. So why include them at all? Because they serve as an excellent way of internally documenting the program. When a program is printed out, those comments appear along with the program instructions.

Comments may appear anywhere, except within a literal, and they may be of any length. That is to say, a comment may start on the same line as another statement or on a line of its own. It may also extend over any number of lines if the terminating delimiter is placed at the appropriate point. Should the delimiter be omitted, the rest of the program would become one long comment!

A word of caution to Pascal programmers: In Pascal, the curly braces are used to delimit comments. Be careful not to fall back into this habit when writing C programs.

If a general comment is required (for example, one describing the role of the entire program), usually it is placed on a line by itself. If the comment refers to a particular line only, it should be included on the same line as the statement, usually to the right of it. It is not unusual to phrase comments in terms of the instructions that one human being would give another.

Comments may not be essential in a simple program, but in more complex situations they are a veritable lifesaver. For any program, no matter how well written, the day will surely arrive when it must be amended in order to improve it, to allow for changing conditions, or to accommodate some governmental directive. The chances are quite high that the original author of the program will not be present when those changes have to be made. Even the original author may not remember the logic of a program that he wrote several months or years before.

In order to lessen the burden on the person who must amend the program, ample comments should be included at the time when the program is being written. Many programmers are tempted to postpone this obligation until the program has been checked out and passed, but then the task of including the comments is not very attractive.

Some programmers feel that C is such a concise language that every line of code should be accompanied by a comment. Too many comments actually can reduce the readability of the program. Even so, it is better to have too many comments than too few.

```
/* a short poem */

main()
{
    printf("Mary had a little lamb, ");
    printf("its fleece was white as snow.\n");
    printf("And everywhere that Mary went, ");
    printf("the lamb was sure to go.\n");
}
```

Program 1-4

```
Mary had a little lamb, its fleece was white as snow.
And everywhere that Mary went, the lamb was sure to go.
```

Output 1-4

Careful attention should be given to the first and third *printf* statements in Program 1-4. Notice that a space has been included at the end of those literals, so that when the lines are printed they will be correctly spaced.

You have now seen some elementary programs. Before long, we shall be examining more interesting ones, but first we have to become acquainted with some of the "nuts and bolts" of the C language. Among the most important of these are variables and the assignment statement, both of which are discussed in the next chapter.

1.3 SAMPLE C PROGRAMS

Before discussing any specific features of C, we shall look at some sample C programs and analyse and understand how they work.

Sample Program 1: Printing a Message

Consider a very simple program given in Fig. 1.1. This program when executed, will produce the following output:

```
I see, I remember
```

```
main( )
{
/*............printing begins...............*/
        printf("I see, I remember");
/*............printing ends ................*/
}
```

Fig. 1.1 *A program to print one line of text*

Let us have a close look at the program. The first line informs the system that the name of the program is **main** and the execution begins at this line. The **main( )** is a special function used by the C system to tell the computer where the program starts. Every program must have *exactly one* **main** function. If we use more than one **main** function, the compiler cannot tell which one marks the beginning of the program.

The empty pair of parentheses immediately following **main** indicates that the function **main** has no *arguments* (or parameters). The concept of arguments will be discussed in detail later when we discuss functions (in Chapter 9).

The opening brace '{' in the second line marks the beginning of the function **main** and the closing brace '}' in the last line indicates the end of the function. In this case, the closing brace also marks the end of the program. All the statements between these two braces form the *function body*. The function body contains a set of instructions to perform the given task.

In this case, the function body contains three statements, out of which only the **printf** line is an executable statement. The lines beginning with /* and ending with */ are known as *comment lines*. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements and therefore anything between /* and */ is ignored by the compiler. In general, a comment can be inserted anywhere blank spaces can occur—at the beginning, middle, or end of a line, but never in the middle of a word.

Because comments do not affect the execution speed and the size of a program, we should use them liberally in our programs. They help the programmers and other users in understanding the various functions and operations of a program and serve as an aid to debugging and testing. We shall see the use of comment lines more in the examples that follow.

Let us now look at the **printf( )** function, the only executable statement of the program.

```
printf("I see, I remember");
```

printf is a predefined, standard C function for printing output. *Predefined* means that it is a function that has already been written and compiled, and linked together with our program at the time of linking. The concepts of compilation and linking are explained later in this chapter (Section 1.6). The **printf** function causes everything between the starting and the ending quotation marks to be printed out. In this case, the output will be:

```
I see, I remember
```

Note that the print line ends with a semicolon. *Every statement in C should end with a semicolon* (;) mark.

Suppose we want to print the above quotation in two lines as

```
I see,
I remember !
```

This can be achieved by adding *another* **printf** function as shown below:

```
printf("I see, \n");
printf("I remember !");
```

The information contained between the parentheses is called the *argument* of the function. The argument of the first **printf** function is "I see,\n" and the second is "I remember !". These arguments are simply strings of characters to be printed out.

Notice that the argument of the first **printf** contains a combination of two characters \ and **n** at the end of the string. This combination is collectively called the *newline* character. A newline character instructs the computer to go to the next (new) line. It is similar in concept to the carriage return key on a typewriter. After printing the character comma (,) the presence of the newline character **\n** causes the string "I remember" to be printed on the next line. No space is allowed between \ and n.

If we omit the newline character from the first **printf** statement, then the output will again be a single line as shown below.

```
I see,I remember !
```

This is similar to the output of the program in Fig. 1.1. However, note that there is no space between , and I.

It is also possible to produce two or more lines of output by one **printf** statement with the use of newline characters at appropriate places. For example, the statement

```
printf("I see,\n I remember !");
```

will output

```
I see,
I remember !
```

while the statement

printf("I\n.see,\n.....I\n......remember !");

will print out

```
I
. see,
.....I
...... remember !
```

Before we proceed to discuss further examples, we must note one important point. C does make a distinction between *uppercase* and *lowercase* letters. For example, **printf** and **PRINTF** are not the same. In C, everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants. We may also use uppercase letters in output strings like "I SEE" and "I REMEMBER".

The above example that printed **I see, I remember** is one of the simplest programs. Figure 1.2 highlights the general format of such simple programs. All C programs need a **main** function.

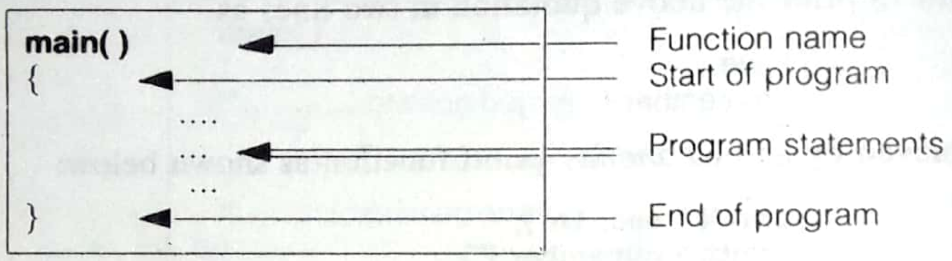

Fig. 1.2 *Format of simple C programs*

Scanned by CamScanner

1.4 BASIC STRUCTURE OF C PROGRAMS

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called *functions*. A function is a subroutine that may include one or more *statements* designed to perform a *specific task*. To write a C program, we first create functions and then put them together. A C program may contain one or more sections shown in Fig. 1.8.

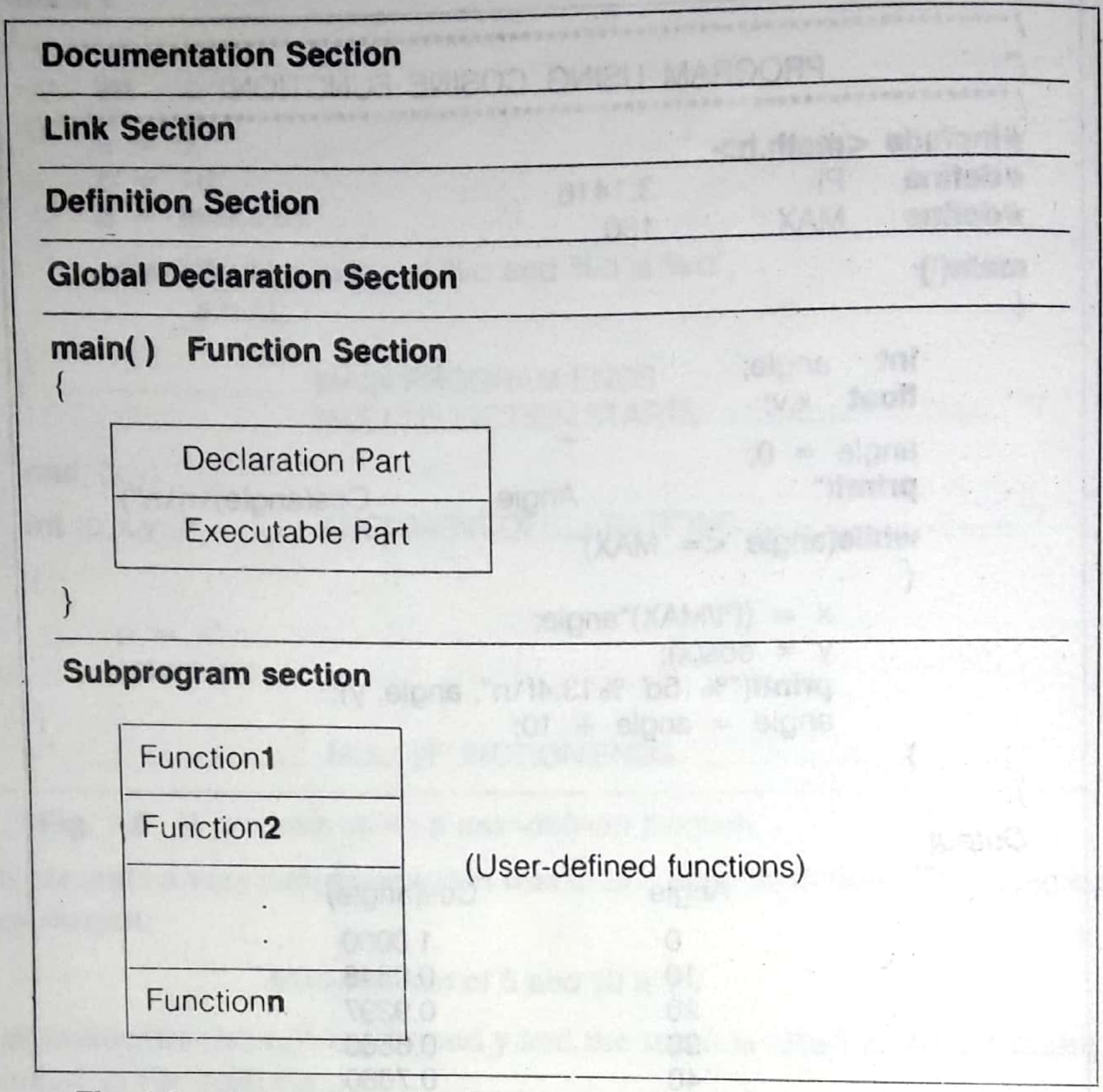

Fig. 1.8 *An overview of a C program*

The documentation section consists of a set of comment lines giving the name of the program, the author and other details which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called *global* variables and are declared in the global declaration section that is outside of all the functions.

Every C program must have one **main( )** function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon.

The subprogram section contains all the user-defined functions that are called in the **main** function. User-defined functions are generally placed immediately after the **main** function, although they may appear in any order.

All sections, except the **main** function section may be absent when they are not required.