Chapter 1

Overview of C

1.1 HISTORY OF C

'C' seems a strange name for a programming language. But this strange sounding language is one of the most popular computer languages today because it is a structured, high-level, machine independent language. It allows software developers to develop programs without worrying about the hardware platforms where they will be implemented.

The root of all modern languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure. Although it never became popular in USA, it was widely used in Europe. ALGOL gave the concept of structured programming to the computer science community. Computer scientists like Corrado Bohm, Guiseppe Jacopini and Edsger Dijkstra popularized this concept during 1960s. Subsequently, several languages were announced.

In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system software. In 1970, Ken Thompson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX operating system at Bell Laboratories. Both BCPL and B were "typeless" system programming languages.

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C. UNIX is one of the most popular network operating systems in use today and the heart of the Internet data superhighway.

For many years, C was used mainly in academic environments, but eventually with the release of many C compilers for commercial use and the increasing popularity of UNIX, it began to gain widespread support among computer professionals. Today, C is running under a variety of operating system and hardware platforms.

During 1970s, C had evolved into what is now known as "traditional C". The language became more popular after publication of the book *'The C Programming Language'* by Brian Kerningham and Dennis Ritchie in 1978. The book was so popular that the language came to be known as "K&R C" among the programming community. The rapid growth of C led to the development of different

versions of the language that were similar but often incompatible. This posed a serious problem for system developers.

To assure that the C language remains standard, in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in 1989 which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. The standard was updated in 1999. The history of ANSI C is illustrated in Fig. 1.1.

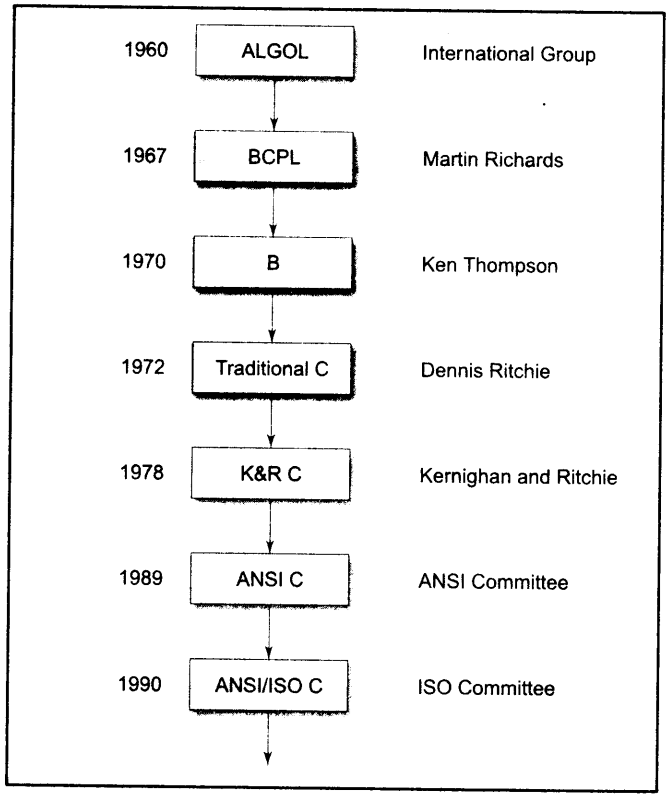

Fig. 1.1 *History of ANSI C*

1.2 IMPORTANCE OF C

The increasing popularity of C is probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of a high-level language and therefore it is well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC. For example, a program to increment a variable from 0 to 15000 takes about one second in C while it takes more than 50 seconds in an interpreter BASIC.

There are only 32 keywords and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.

C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C language is well suited for structured programming, thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance easier.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to C library. With the availability of a large number of functions, the programming task becomes simple.

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

1.3 SAMPLE PROGRAM 1: PRINTING A MESSAGE

Consider a very simple program given in Fig. 1.2.

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

Fig. 1.2 *A program to print one line of text*

This program when executed will produce the following output:

```
I see, I remember
```

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 wherever blank spaces can occur—at the beginning, middle or end of a line—"but never in the middle of a word ".

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The comment line

```
/* = = = =/* = = = = */ = = = = */
```

is not valid and therefore results in an error.

Since comments do not affect the execution speed and the size of a compiled 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. 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. This 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. 1.2. 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 character 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 !
```

NOTE: Some authors recommend the inclusion of the statement

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

at the beginning of all programs that use any input/output library functions. However, this is not necessary for the functions *printf* and *scanf* which have been defined as a part of the C language. See Chapter 4 for more on input and output functions.

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 1.3 highlights the general format of such simple programs. All C programs need a **main** function.

The main Function

The main is a part of every C program. C permits different forms of main statement. Following forms are allowed.

- main()
- int main()
- void main()
- main(void)
- void main(void)
- int main(void)

The empty pair of parentheses indicates that the function has no arguments. This may be explicitly indicated by using the keyword **void** inside the parentheses. We may also specify the keyword **int** or **void** before the word **main**. The keyword **void** means that the function does not return any information to the operating system and **int** means that the function returns an integer value to the operating system. When **int** is specified, the last statement in the program must be "return 0". For the sake of simplicity, we use the first form in our programs.

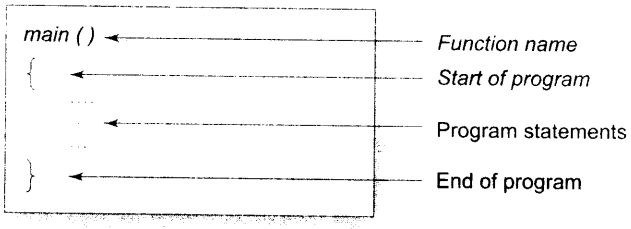

Fig. 1.3 *Format of simple C programs*

1.4 SAMPLE PROGRAM 2: ADDING TWO NUMBERS

Consider another program, which performs addition on two numbers and displays the result. The complete program is shown in Fig. 1.4.

```
/* Programm ADDITION                                    line-1 */
/* Written by EBG                                       line-2 */
main()                                                  /* line-3 */
{                                                       /* line-4 */
          int number;                                   /* line-5 */
          float amount;                                 /* line-6 */
                                                        /* line-7 */
          number = 100;                                 /* line-8 */
                                                        /* line-9 */
          amount = 30.75 + 75.35;                       /* line-10 */
          printf("%d\n",number);                        /* line-11 */
          printf("%5.2f",amount);                       /* line-12 */
}                                                       /* line-13 */
```

Fig. 1.4 *Program to add two numbers*

This program when executed will produce the following output:

```
100
106.10
```

The first two lines of the program are comment lines. It is a good practice to use comment lines in the beginning to give information such as name of the program, author, date, etc. Comment characters are also used in other lines to indicate line numbers.

The words **number** and **amount** are *variable names* that are used to store numeric data. The numeric data may be either in *integer* form or in *real* form. In C, *all variables should be declared* to tell the compiler what the ***variable names*** are and what ***type of data*** they hold. The variables must be declared before they are used. In lines 5 and 6, the declarations

```
int number;
float amount;
```

tell the compiler that **number** is an integer (**int** is the abbreviation for integer) and **amount** is a floating (**float**) point number. Declaration statements must appear at the beginning of the functions as shown in Fig.1.4. All declaration statements end with a semicolon. C supports many other data types and they are discussed in detail in Chapter 2.

The words such as **int** and **float** are called the *keywords* and cannot be used as *variable* names. A list of keywords is given in Chapter 2.

Data is stored in a variable by *assigning* a data value to it. This is done in lines 8 and 10. In line-8, an integer value 100 is assigned to the integer variable **number** and in line-10, the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable **amount.** The statements

```
number = 100;
amount = 30.75 + 75.35;
```

are called the *assignment* statements. Every assignment statement must have a semicolon at the end.

The next statement is an output statement that prints the value of **number**. The print statement

```
printf("%d\n", number);
```

contains two arguments. The first argument "%d" tells the compiler that the value of the second argument **number** should be printed as a *decimal integer*. Note that these arguments are separated by a comma. The newline character \n causes the next output to appear on a new line.

The last statement of the program

```
printf("%5.2f", amount);
```

prints out the value of **amount** in floating point format. The format specification *%5.2f* tells the compiler that the output must be in *floating point*, with five places in all and two places to the right of the decimal point.

1.5 SAMPLE PROGRAM 3: INTEREST CALCULATION

The program in Fig. 1.5 calculates the value of money at the end of each year of investment, assuming an interest rate of 11 percent and prints the year, and the corresponding amount, in two columns. The output is shown in Fig. 1.6 for a period of 10 years with an initial investment of 5000.00. The program uses the following formula:

Value at the end of year = Value at start of year (1 + interest rate)

In the program, the variable **value** represents the value of money at the end of the year while **amount** represents the value of money at the start of the year. The statement

```
amount = value ;
```

makes the value at the end of the *current* year as the value at start of the *next* year.

```
/*---------- INVESTMENT PROBLEM ----------*/
#define PERIOD  10
#define PRINCIPAL  5000.00
/*---------- MAIN PROGRAM BEGINS ----------*/
main()
```

```
{ /*———————— DECLARATION STATEMENTS ————————*/
    int year;
    float amount, value, inrate;
/*———————— ASSIGNMENT STATEMENTS ————————*/
    amount = PRINCIPAL;
    inrate = 0.11;
    year = 0;
/*———————— COMPUTATION STATEMENTS ————————*/
/*———————— COMPUTATION USING While LOOP ————————*/
    while(year <= PERIOD)
    {   printf("%2d        %8.2f\n",year, amount);
        value  = amount + inrate * amount;
        year   = year + 1;
        amount = value;
    }
/*———————— while LOOP ENDS ————————*/
}
/*———————— PROGRAM ENDS ————————*/
```

Fig. 1.5 *Program for investment problem*

Let us consider the new features introduced in this program. The second and third lines begin with **#define** instructions. A **#define** instruction defines value to a *symbolic constant* for use in the program. Whenever a symbolic name is encountered, the compiler substitutes the value associated with the name automatically. To change the value, we have to simply change the definition. In this example, we have defined two symbolic constants **PERIOD** and **PRINCIPAL** and assigned values 10 and 5000.00 respectively. These values remain constant throughout the execution of the program.

```
0        5000.00
1        5550.00
2        6160.50
3        6838.15
4        7590.35
5        8425.29
6        9352.07
7        10380.00
8        11522.69
9        12790.00
10       14197.11
```

Fig. 1.6 *Output of the investment program*

The **#define** Directive

A **#define** is a preprocessor compiler directive and not a statement. Therefore **#define** lines should not end with a semicolon. Symbolic constants are generally written in uppercase so that they are easily distinguished from lowercase variable names. **#define** instructions are usually placed at the beginning before the **main()** function. Symbolic constants are not declared in declaration section. Preprocessor directions are discussed in Chapter 14.

We must note that the defined constants are not variables. We may not change their values within the program by using an assignment statement. For example, the statement

```
PRINCIPAL = 10000.00;
```

is illegal.

The declaration section declares **year** as integer and **amount, value** and **inrate** as floating point numbers. Note all the floating-point variables are declared in one statement. They can also be declared as

```
float amount;
float value;
float inrate;
```

When two or more variables are declared in one statement, they are separated by a comma.

All computations and printing are accomplished in a **while** loop. **while** is a mechanism for evaluating repeatedly a statement or a group of statements. In this case as long as the value of **year** is less than or equal to the value of **PERIOD**, the four statements that follow **while** are executed. Note that these four statements are grouped by braces. We exit the loop when **year** becomes greater than **PERIOD**. The concept and types of loops are discussed in Chapter 6.

C supports the basic four arithmetic operators (–, +, *, /) along with several others. They are discussed in Chapter 3.

1.6 SAMPLE PROGRAM 4: USE OF SUBROUTINES

So far, we have used only **printf** function that has been provided for us by the C system. The program shown in Fig. 1.7 uses a user-defined function. A function defined by the user is equivalent to a subroutine in FORTRAN or subprogram in BASIC.

```
/*———————— PROGRAM USING FUNCTION ————————*/
int mul (int a, int b); /*—— DECLARATION ———*/
/*————————— MAIN PROGRAM BEGINS ——————————*/
     main ()
     {
          int a, b, c;

          a = 5;
```

```
        b = 10;
        c = mul (a,b);

        printf ("multiplication of %d and %d is %d",a,b,c);
    }
/* ————————      MAIN PROGRAM ENDS
                  MUL() FUNCTION STARTS ————————————*/
    int mul (int x, int y)
    int p;
    {
          p = x*y;
          return(p);
    }
/* ——————————— MUL () FUNCTION ENDS ————————————*/
```

Fig. 1.7 *A program using a user-defined function*

Figure 1.7 presents a very simple program that uses a **mul** ( ) function. The program will print the following output.

Multiplication of 5 and 10 is 50

The **mul** ( ) function multiplies the values of **x** and **y** and the result is returned to the **main** ( ) function when it is called in the statement

```
c = mul (a, b);
```

The **mul** ( ) has two *arguments* **x** and **y** that are declared as integers. The values of **a** and **b** are passed on to **x** and **y** respectively when the function **mul** ( ) is called. User-defined functions are considered in detail in Chapter 9.

1.7 SAMPLE PROGRAM 5: USE OF MATH FUNCTIONS

We often use standard mathematical functions such as cos, sin, exp, etc. We shall see now the use of a mathematical function in a program. The standard mathematical functions are defined and kept as a part of C **math library**. If we want to use any of these mathematical functions, we must add an **#include** instruction in the program. Like **#define,** it is also a compiler directive that instructs the compiler to link the specified mathematical functions from the library. The instruction is of the form

```
#include <math.h>
```

math.h is the filename containing the required function. Figure 1.8 illustrates the use of cosine function. The program calculates cosine values for angles 0, 10, 20.....180 and prints out the results with headings.

Another **#include** instruction that is often required is

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

stdio.h refers to the *standard* I/O header file containing standard input and output functions

```
/*------- PROGRAM USING COSINE FUNCTION -------*/
#include <math.h>
#define    PI     3.1416
#define    MAX    180

main ( )
{

     int angle;
     float x,y;

     angle = 0;
     printf("    Angle    Cos(angle)\n\n");

     while(angle <= MAX)
     {
          x = (PI/MAX)*angle;
          y = cos(x);
          printf("%15d %13.4f\n", angle, y);
          angle = angle + 10;
     }
}
```

Output

Angle	Cos(angle)
0	1.0000
10	0.9848
20	0.9397
30	0.8660
40	0.7660
50	0.6428
60	0.5000
70	0.3420
80	0.1736
90	-0.0000
100	-0.1737
110	-0.3420
120	-0.5000
130	-0.6428
140	-0.7660
150	-0.8660
160	-0.9397
170	-0.9848
180	-1.0000

Fig. 1.8 *Program using a math function*

The #include Directive

As mentioned earlier, C programs are divided into modules or functions. Some functions are written by users like us and many others are stored in the C library. Library functions are grouped category-wise and stored in different files known as header files. If we want to access the functions stored in the library, it is necessary to tell the compiler about the files to be accessed.

This is achieved by using the preprocessor directive **#include** as follows:

#include<filename>

filename is the name of the library file that contains the required function definition. Preprocessor directives are placed at the beginning of a program.

A list of library functions and header files containing them are given in Appendix III.

1.8 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. 1.9.

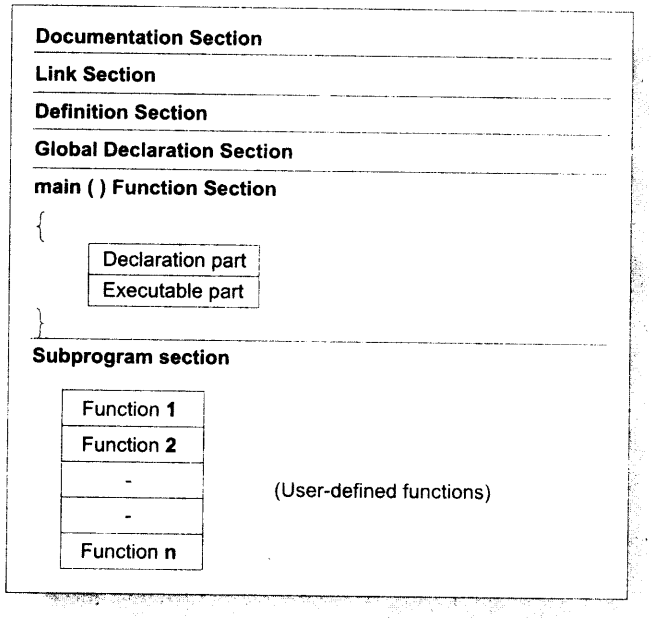

Fig. 1.9 *An overview of a C program*

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. This section also declares all the user-defined 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.

1.9 PROGRAMMING STYLE

Unlike some other programming languages (COBOL, FORTRAN, etc.,) C is a *free-form* language. That is, the C compiler does not care, where on the line we begin typing. While this may be a licence for bad programming, we should try to use this fact to our advantage in developing readable programs. Although several alternative styles are possible, we should select one style and use it with total consistency.

First of all, we must develop the habit of writing programs in lowercase letters. C program statements are written in lowercase letters. Uppercase letters are used only for symbolic constants.

Braces group program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug. Note how the braces are aligned and the statements are indented in the program of Fig. 1.5.

Since C is a free-form language, we can group statements together on one line. The statements

```
a = b;
x = y + 1;
z = a + x;
```

can be written on one line as

```
a = b; x = y+1; z = a+x;
```

The program

```
main( )
{
    printf("hello C");
}
```

may be written in one line like

```
main( ) {printf("Hello C")};
```

However, this style make the program more difficult to understand and should not be used. In this book, each statement is written on a separate line.

The generous use of comments inside a program cannot be overemphasized. Judiciously inserted comments not only increase the readability but also help to understand the program logic. This is very important for debugging and testing the program.

1.10 EXECUTING A 'C' PROGRAM

Executing a program written in C involves a series of steps. These are:

1. Creating the program
2. Compiling the program
3. Linking the program with functions that are needed from the C library
4. Executing the program.

Figure 1.10 illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the *operating system*, system commands for implementing the steps and conventions for naming *files* may differ on different systems.

An operating system is a program that controls the entire operation of a computer system. All input/out operations are channeled through the operating system. The operating system, which is an interface between the hardware and the user, handles the execution of user programs.

The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (for microcomputers). We shall discuss briefly the procedure to be followed in executing C programs under both these operating systems in the following sections.

1.11 UNIX SYSTEM

Creating the program

Once we load the UNIX operating system into the memory, the computer is ready to receive program. The program must be entered into a file. The file name can consist of letters, digits and special characters, followed by a dot and a letter **c**. Examples of valid file names are:

hello.c

program.c

ebg1.c

The file is created with the help of a *text editor*, either **ed** or **vi.** The command for calling the editor and creating the file is

ed *filename*

If the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor. (The name of your system's editor may be different. Check your system manual.)

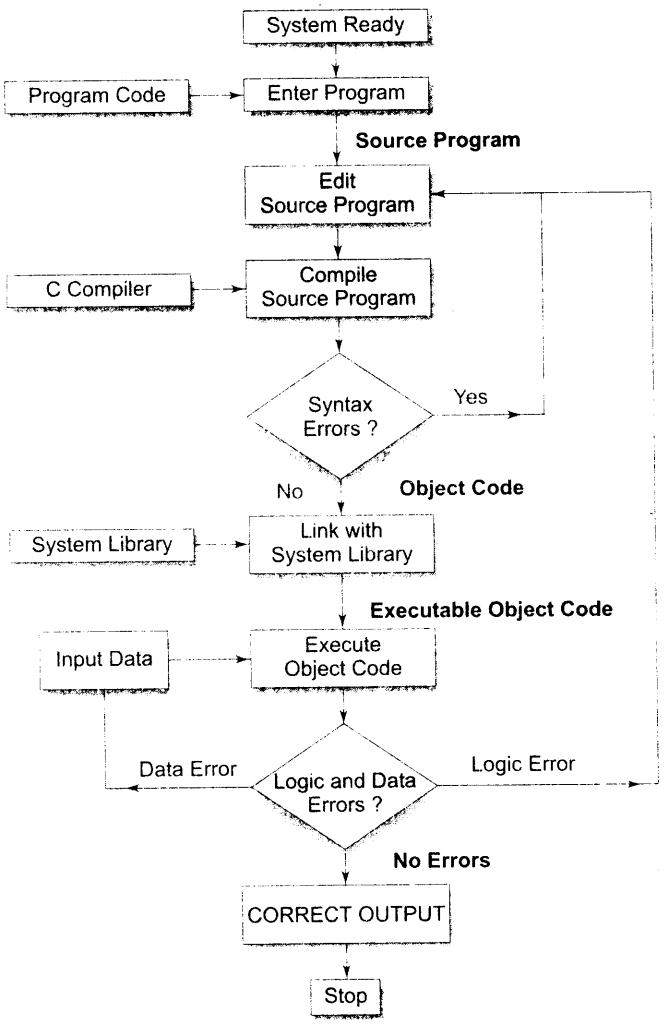

Fig. 1.10 *Process of compiling and running a C program*

When the editing is over, the file is saved on disk. It can then be referenced any time later by its file name. The program that is entered into the file is known as the *source program*, since it represents the original form of the program.

Compiling and Linking

Let us assume that the source program has been created in a file named *ebg1.c*. Now the program is ready for compilation. The compilation command to achieve this task under UNIX is

```
cc ebg1.c
```

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is

alright, the compilation proceeds silently and the translated program is stored on another file with the name *ebg1.o*. This program is known as *object code*.

Linking is the process of putting together other program files and functions that are required by the program. For example, if the program is using **exp()** function, then the object code of this function should be brought from the **math library** of the system and linked to the main program. Under UNIX, the linking is automatically done (if no errors are detected) when the **cc** command is used.

If any mistakes in the *syntax* and *semantics* of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again.

The compiled and linked program is called the *executable object code* and is stored automatically in another file named **a.out.**

Note that some systems use different compilation command for linking mathematical functions.

```
cc filename - lm
```

is the command under UNIPLUS SYSTEM V operating system.

Executing the Program

Execution is a simple task. The command

```
a.out
```

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produce the desired results. Perhaps, something is wrong with the program *logic* or *data*. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

Creating Your Own Executable File

Note that the linker always assigns the same name **a.out**. When we compile another program, this file will be overwritten by the executable object code of the new program. If we want to prevent from happening, we should rename the file immediately by using the command.

```
mv a.out name
```

We may also achieve this by specifying an option in the cc command as follows:

```
cc -o name source-file
```

This will store the executable object code in the file name and prevent the old file **a.out** from being destroyed.

Multiple Source Files

To compile and link multiple source program files, we must append all the files names to the **cc** command.

```
cc filename-1.c .... filename-n.c
```

These files will be separately compiled into object files called

```
filename-i.o
```

and then linked to produce an executable program file **a.out** as shown in Fig. 1.11.

It is also possible to compile each file separately and link them later. For example, the commands

```
cc -c mod1.c
cc -c mod2.c
```

will compile the source files *mod1.c* and *mod2.c* into objects files *mod1.o* and *mod2.o*. They can be linked together by the command

```
cc mod1.o mod2.o
```

we may also combine the source files and object files as follows:

```
cc mod1.c mod2.o
```

Only *mod1.c* is compiled and then linked with the object file mod2.o. This approach is useful when one of the multiple source files need to be changed and recompiled or an already existing object files is to be used along with the program to be compiled.

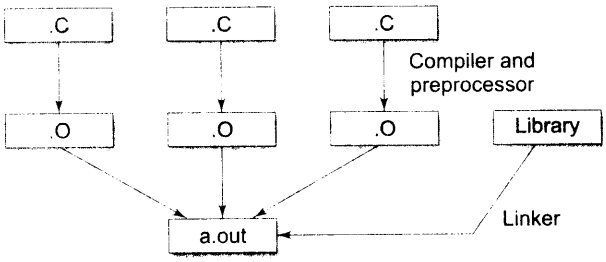

Fig. 1.11 *Compilation of multiple files*

1.12 MS-DOS SYSTEM

The program can be created using any word processing software in non-document mode. The file name should end with the characters ".c" like **program.c, pay.c,** etc. Then the command

```
MSC pay.c
```

under MS-DOS operating system would load the program stored in the file **pay.c** and generate the **object code.** This code is stored in another file under name **pay.obj**. In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again.

The linking is done by the command

```
LINK pay.obj
```

which generates the **executable code** with the filename **pay.exe**. Now the command

```
pay
```

would execute the program and give the results.

Just Remember

- Every C program requires a **main()** function (Use of more than one **main()** is illegal). The place **main** is where the program execution begins.
- The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace.
- C programs are written in lowercase letters. However, uppercase letters are used for symbolic names and output strings.
- All the words in a program line must be separated from each other by at least one space, or a tab, or a punctuation mark.
- Every program statement in a C language must end with a semicolon.
- All variables must be declared for their types before they are used in the program.
- We must make sure to include header files using **#include** directive when the program refers to special names and functions that it does not define.
- Compiler directives such as **define** and **include** are special instructions to the compiler to help it compile a program. They do not end with a semicolon.
- The sign # of compiler directives must appear in the first column of the line.
- When braces are used to group statements, make sure that the opening brace has a corresponding closing brace.
- C is a free-form language and therefore a proper form of indentation of various sections would improve legibility of the program.
- A comment can be inserted almost anywhere a space can appear. Use of appropriate comments in proper places increases readability and understandability of the program and helps users in debugging and testing. Remember to match the symbols /* and */ appropriately.

REVIEW QUESTIONS

1.1 State whether the following statements are *true* or *false*.
- (a) Every line in a C program should end with a semicolon.
- (b) In C language lowercase letters are significant.
- (c) Every C program ends with an END word.
- (d) **main( )** is where the program begins its execution.
- (e) A line in a program may have more than one statement.
- (f) A **printf** statement can generate only one line of output.
- (g) The closing brace of the **main( )** in a program is the logical end of the program.
- (h) The purpose of the header file such as **stdio.h** is to store the source code of a program.
- (i) Comments cause the computer to print the text enclosed between /* and */ when executed.
- (j) Syntax errors will be detected by the compiler.

1.2 Which of the following statements are *true*?
- (a) Every C program must have at least one user-defined function.

(b) Only one function may be named **main( )**.
(c) Declaration section contains instructions to the computer.

1.3 Which of the following statements about comments are *false*?
(a) Use of comments reduces the speed of execution of a program.
(b) Comments serve as internal documentation for programmers.
(c) A comment can be inserted in the middle of a statement.
(d) In C, we can have comments inside comments.

1.4 Fill in the blanks with appropriate words in each of the following statements.
(a) Every program statement in a C program must end with a _____
(b) The _____ Function is used to display the output on the screen.
(c) The _____ header file contains mathematical functions.
(d) The escape sequence character _____ causes the cursor to move to the next line on the screen.

1.5 Remove the semicolon at the end of the **printf** statement in the program of Fig. 1.2 and execute it. What is the output?

1.6 In the Sample Program 2, delete line-5 and execute the program. How helpful is the error message?

1.7 Modify the Sample Program 3 to display the following output:

Year	**Amount**
1	5500.00
2	6160.00
.	_____
.	_____
10	14197.11

1.8 Find errors, if any, in the following program:

```
/* A simple program
int main( )
{
     /* Does nothing */
}
```

1.9 Find errors, if any, in the following program:

```
#include (stdio.h)
void main(void)
{
     print("Hello C");
}
```

1.10 Find errors, if any, in the following program:

```
Include <math.h>
main { }
(
     FLOAT X;
     X = 2.5;
     Y = exp(x);
```

```
        Print(x,y);
}
```

1.11 Why and when do we use the **#define** directive?
1.12 Why and when do we use the **#include** directive?
1.13 What does **void main(void)** mean?
1.14 Distinguish between the following pairs:
(a) main( ) and void main(void)
(b) int main( ) and void main( )
1.15 Why do we need to use comments in programs?
1.16 Why is the look of a program is important?
1.17 Where are blank spaces permitted in a C program?
1.18 Describe the structure of a C program.
1.19 Describe the process of creating and executing a C program under UNIX system.
1.20 How do we implement multiple source program files?

PROGRAMMING EXERCISES

1.1 Write a program that will print your mailing address in the following form:

First line	:	Name
Second line	:	Door No, Street
Third line	:	City, Pin code

1.2 Modify the above program to provide border lines to the address.
1.3 Write a program using one print statement to print the pattern of asterisks as shown below:

```
*
*   *
*   *   *
*   *   *   *
```

1.4 Write a program that will print the following figure using suitable characters.

1.5 Given the radius of a circle, write a program to compute and display its area. Use a symbolic constant to define the π value and assume a suitable value for radius.
1.6 Write a program to output the following multiplication table:

$5 \times 1 = 5$
$5 \times 2 = 10$
$5 \times 3 = 15$
. .
. .
$5 \times 10 = 50$

1.7 Given two integers 20 and 10, write a program that uses a function add( ) to add these two numbers and sub( ) to find the difference of these two numbers and then display the sum and difference in the following form:

$$20 + 10 = 30$$

$$20 - 10 = 10$$

1.8 Given the values of three variables a, b and c, write a program to compute and display the value of x, where

$$x = \frac{a}{b - c}$$

Execute your program for the following values:

(a) a = 250, b = 85, c = 25

(b) a = 300, b = 70, c = 70

Comment on the output in each case.

Chapter 2

Constants, Variables, and Data Types

2.1 INTRODUCTION

A programming language is designed to help process certain kinds of *data* consisting of numbers, characters and strings and to provide useful output known as *information*. The task of processing of data is accomplished by executing a sequence of precise instructions called a *program*. These instructions are formed using certain symbols and words according to some rigid rules known as *syntax rules* (or *grammar*). Every program instruction must confirm precisely to the syntax rules of the language.

Like any other language, C has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to C programming language.

2.2 CHARACTER SET

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. However, a subset of characters is available that can be used on most personal, micro, mini and mainframe computers. The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special characters
4. White spaces

The entire character set is given in Table 2.1.

The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

Trigraph Characters

Many non-English keyboards do not support all the characters mentioned in Table 2.1. ANSI C introduces the concept of "trigraph" sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in Table 2.2. For example, if a keyboard does not support square brackets, we can still use them in a program using the trigraphs ??(and ??).

Table 2.1 *C Character Set*

Letters		*Digits*
Uppercase A.....Z		All decimal digits 0 9
Lowercase a.....z		
	Special Characters	
, comma		& ampersand
. period		^ caret
; semicolon		* asterisk
: colon		- minus sign
? question mark		+ plus sign
' apostrophe		< opening angle bracket
" quotation mark		(or less than sign)
! exclamation mark		> closing angle bracket
| vertical bar		(or greater than sign)
/ slash		(left parenthesis
\ backslash		) right parenthesis
~ tilde		[left bracket
_ under score		] right bracket
$ dollar sign		{ left brace
% percent sign		} right brace
		# number sign
	White Spaces	
	Blank space	
	Horizontal tab	
	Carriage return	
	New line	
	Form feed	

Table 2.2 *ANSI C Trigraph Sequences*

Trigraph sequence	*Translation*
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	| vetical bar
??/	\ back slash
??'	^ caret
??-	~ tilde

2.3 C TOKENS

In a passage of text, individual words and punctuation marks are called *tokens*. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens as shown in Fig. 2.1. C programs are written using these tokens and the syntax of the language.

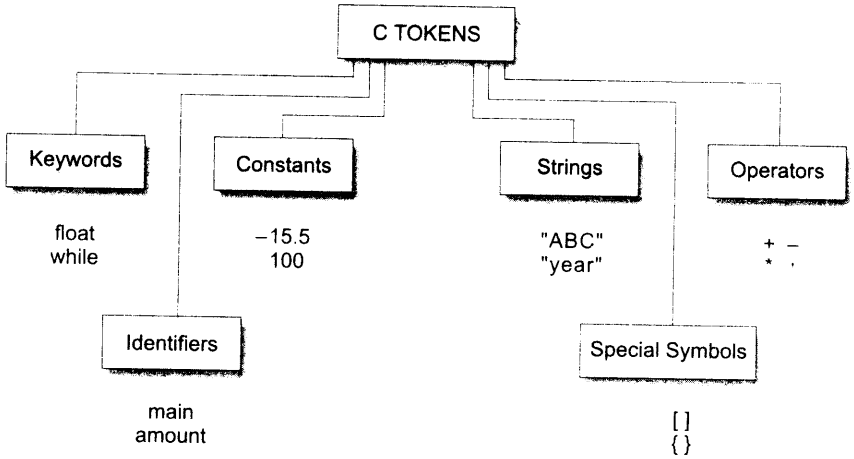

Fig. 2.1 *C tokens and examples*

2.4 KEYWORDS AND IDENTIFIERS

Every C word is classified as either a *keyword* or an *identifier*. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords of ANSI C are listed in Table 2.3. All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

Table 2.3 *ANSI C Keywords*

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

Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.

Rules for Identifiers

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space.

2.5 CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in Fig. 2.2.

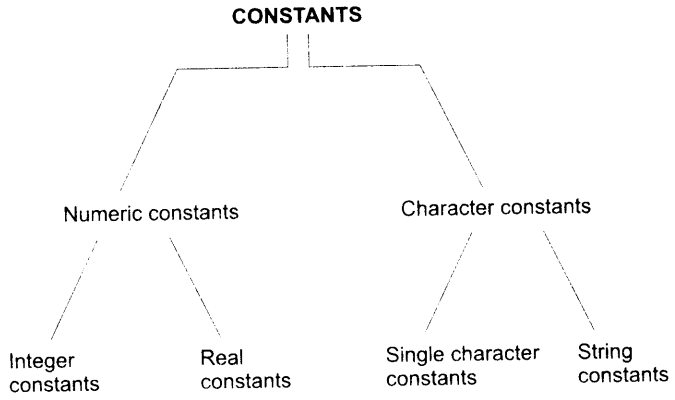

Fig. 2.2 *Basic types of C constants*

Integer Constants

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, *decimal* integer, *octal* integer and *hexadecimal* integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are:

123 – 321 0 654321 +78

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

15 750 20,000 $1000

are illegal numbers. Note that ANSI C supports *unary plus* which was not defined earlier.

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

037 0 0435 0551

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f. The letter A through F represent the numbers 10 through 15. Following are the examples of valid hex integers.

0X2 0x9F 0Xbcd 0x

We rarely use octal and hexadecimal numbers in programming.

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending *qualifiers* such as U,L and UL to the constants. For examples:

56789U or 56789u (unsigned integer)
987612347UL or 98761234ul (unsigned long integer)
9876543L or 98765431 (long integer)

The concept of unsigned and long integers are discussed in detail in Section 2.7.

Example 2.1 Representation of integer constants on a 16-bit computer.

The program in Fig.2.3 illustrates the use of integer constants on a 16-bit machine. The output in Fig. 2.3 shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

```
Program
    main()
    {
        printf("Integer values\n\n");
        printf("%d %d %d\n", 32767,32767+1,32767+10);
        printf("\n");
        printf("Long integer values\n\n");
        printf("%ld %ld %ld\n", 32767L,32767L+1L,32767L+10L);
    }
Output
    Integer values
    32767 -32768 -32759
    Long integer values
    32767 32768 32777
```

Fig. 2.3 *Representation of integer constants on 16-bit machine*

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing

fractional parts like 17.548. Such numbers are called *real* (or *floating point*) constants. Further examples of real constants are:

0.0083 -0.75 435.36 +247.0

These numbers are shown in *decimal notation*, having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point, or digits after the decimal point. That is,

215. .95 -.71 +.5

are all valid real numbers.

A real number may also be expressed in *exponential* (or *scientific) notation*. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2. The general form is:

mantissa e exponent

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The *exponent* is an integer number with an optional *plus* or *minus sign*. The letter **e** separating the mantissa and the exponent can be written in either lowercase or uppercase. Since the exponent causes the decimal point to "float", this notation is said to represent a real number in *floating point form*. Examples of legal floating-point constants are:

0.65e4 12e-2 1.5e+5 3.18E3 -1.2E-1

Embedded white space is not allowed.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, -0.000000368 is equivalent to -3.68E-7.

Floating-point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single-precision and l or L to extend double precision further.

Some examples of valid and invalid numeric constants are given in Table 2.4.

Table 2.4 *Examples of Numeric Constants*

Constant	*Valid ?*	*Remarks*
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
$255	No	$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of *single* quote marks. Example of character constants are:

'5' 'X' ';' ' '

Note that the character constant '5' is not the same as the *number 5*. The last constant is a blank space.

Character constants have integer values known as ASCII values. For example, the statement

```
printf("%d", 'a');
```

would print the number 97, the ASCII value of the letter a. Similarly, the statement

```
printf("%c", '97');
```

would output the letter 'a'. ASCII values for all characters are given in Appendix II.

Since each character constant represents an integer value, it is also possible to perform arithmetic operations on character constants. They are discussed in Chapter 8.

String Constants

A string constant is a sequence of characters enclosed in *double* quotes. The characters may be letters, numbers, special characters and blank space. Examples are:

"Hello!" "1987" "WELL DONE" "?...!" "5+3" "X"

Remember that a character constant (e.g., 'X') is not equivalent to the single character string constant (e.g., "X"). Further, a single character string constant does not have an equivalent integer value while a character constant has an integer value. Character strings are often used in programs to build meaningful programs. Manipulation of character strings are considered in detail in Chapter 8.

Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 2.5. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known as *escape sequences*.

Table 2.5 *Backslash Character Constants*

Constant	*Meaning*
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab

(Contd.)

Table 2.5 *(Contd.)*

Constant	*Meaning*
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

2.6 VARIABLES

A *variable* is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. In Chapter 1, we used several variables. For instance, we used the variable **amount** in Sample Program 3 to store the value of money at the end of each year (after adding the interest earned during that year).

A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples of such names are:

Average
height
Total
Counter_1
class_strength

As mentioned earlier, variable names may consist of letters, digits, and the underscore(_) character, subject to the following conditions:

1. They must begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognizes a length of 31 characters. However, length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers.
3. Uppercase and lowercase are significant. That is, the varible **Total** is not the same as **total** or **TOTAL**.
4. It should not be a keyword.
5. White space is not allowed.

Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
mark	sum1	distance

Invalid examples include:

123	(area)
%	25th

Further examples of variable names and their correctness are given in Table 2.6.

Table 2.6 *Examples of Variable Names*

Variable name	*Valid ?*	*Remark*
First_tag	Valid	
char	Not valid	char is a keyword
Price$	Not valid	Dollar sign is illegal
group one	Not valid	Blank space is not permitted
average_number	Valid	First eight characters are significant
int_type	Valid	Keyword may be part of a name

If only the first eight characters are recognized by a compiler, then the two names

average_height
average_weight

mean the same thing to the computer. Such names can be rewritten as

avg_height and avg_weight

or

ht_average and wt_average

without changing their meanings.

2.7 DATA TYPES

C language is rich in its *data types*. Storage representations and machine instructions to handle constants differ from machine to machine. The variety of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

ANSI C supports three classes of data types:

1. Primary (or fundamental) data types
2. Derived data types
3. User-defined data types

The primary data types and their extensions are discussed in this section. The user-defined data types are defined in the next section while the derived data types such as arrays, functions, structures and pointers are discussed as and when they are encountered.

All C compilers support five fundamental data types, namely integer **(int)**, character **(char)**, floating point **(float)**, double-precision floating point **(double)** and **void**. Many of them also offer extended data types such as **long int** and **long double**. Various data types and the terminology used to describe them are given in Fig. 2.4. The range of the basic four types are given in Table 2.7. We discuss briefly each one of them in this section.

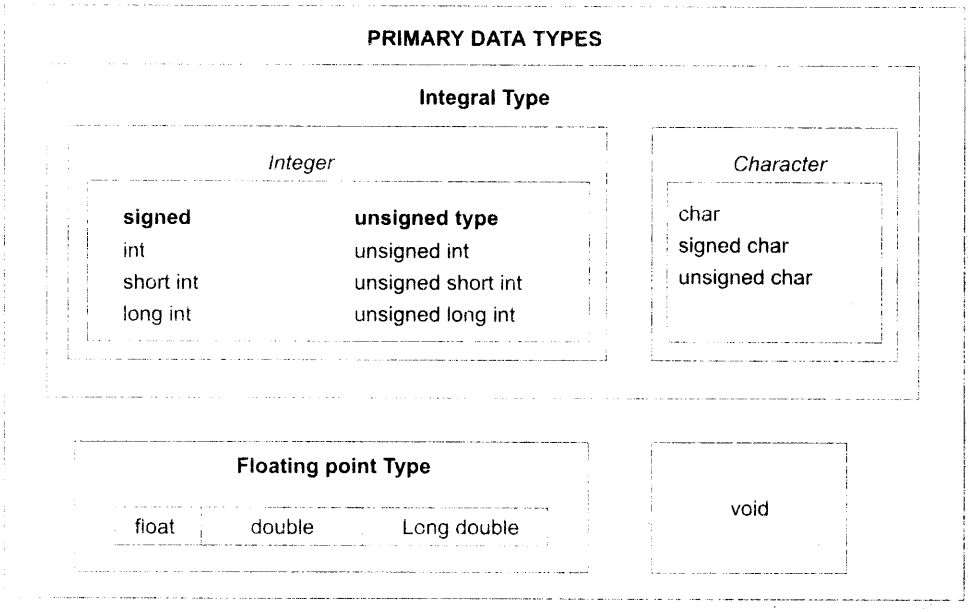

Fig. 2.4 *Primary data types in C*

Table 2.7 *Size and Range of Basic Data Types on 16-bit Machines*

Data type	*Range of values*
char	-128 to 127
int	-32,768 to 32,767
float	3.4e-38 to 3.4e+e38
double	1.7e-308 to 1.7e+308

Integer Types

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to the range -32768 to +32767 (that is, -2^{15} to $+2^{15}-1$). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int,** and **long int,** in both **signed** and **unsigned** forms. ANSI C defines these types so that they can be organized from the smallest to the largest, as shown in Fig. 2.5. For example, **short int** represents fairly small integer values and requires half the amount of storage as a regular **int** number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

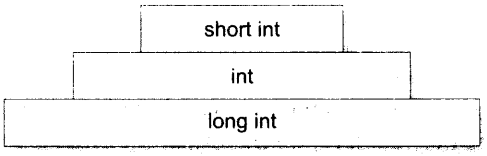

Fig. 2.5 *Integer types*

We declare **long** and **unsigned** integers to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number. Table 2.8 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

Table 2.8 *Size and Range of Data Types on a 16-bit Machine*

Type	*Size (bits)*	*Range*
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int or signed int	16	-32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E - 38 to 3.4E + 38
double	64	1.7E - 308 to 1.7E + 308
long double	80	3.4E - 4932 to 1.1E + 4932

Floating Point Types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword **float.** When the accuracy provided by a **float** number is not sufficient, the type **double** can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits. These are known as *double precision* numbers. Remember that double type represents the same data type that **float** represents, but with a greater precision. To extend the precision further, we may use **long double** which uses 80 bits. The relationship among floating types is illustrated in Fig. 2.6.

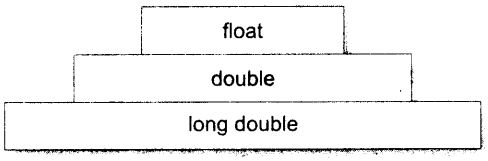

Fig. 2.6 *Floating-point types*

Void Types

The **void** type has no values. This is usually used to specify the type of functions. The type of a function is said to be **void** when it does not return any value to the calling function. It can also play the role of a generic type, meaning that it can represent any of the other standard types.

Character Types

A single character can be defined as a character(**char**) type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier **signed** or **unsigned** may be explicitly applied to char. While **unsigned chars** have values between 0 and 255, **signed chars** have values from –128 to 127.

2.8 DECLARATION OF VARIABLES

After designing suitable variable names, we must declare them to the compiler. Declaration does two things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program.

Primary Type Declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with its type. The syntax for declaring a variable is as follows:

```
data-type v1,v2,....vn ;
```

v1, v2, ...vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. For example, valid declarations are:

```
int count;
int number, total;
double ratio;
```

int and **double** are the keywords to represent integer type and real type data values respectively. Table 2.9 shows various data types and their keyword equivalents.

Table 2.9 *Data Types and Their Keywords*

Data type	*Keyword equivalent*
Character	char
Unsigned character	unsigned char
Signed character	signed char
Signed integer	signed int (or int)

(Contd.)

Data type	*Keyword equivalent*
Signed short integer	signed short int (or short int or short)
Signed long integer	signed long int (or long int or long)
Unsigned integer	unsigned int (or unsigned)
Unsigned short integer	unsigned short int (or unsigned short)
Unsigned long integer	unsigned long int (or unsigned long)
Floating point	float
Double-precision floating point	double
Extended double-precision floating point	long double

The program segment given in Fig. 2.7 illustrates declaration of variables. **main()** is the beginning of the program. The opening brace { signals the execution of the program. Declaration of variables is usually done immediately after the opening brace of the program. The variables can also be declared outside (either before or after) the **main** function. The importance of place of declaration will be dealt in detail later while discussing functions.

```
main() /*.........Program Name....................... */
{
  /*...............Declaration.....................*/
    float       x, y;
    int         code;
    short int   count;
    long int    amount;
    double      deviation;
    unsigned    n;
    char        c;
  /*...............Computation..................... */
    . . . .
    . . . .
    . . . .
} /*............Program ends.....................*/
```

Fig. 2.7 *Declaration of variables*

When an adjective (qualifier) **short, long, or unsigned** is used without a basic data type specifier, C compilers treat the data type as an **int.** If we want to declare a character variable as unsigned, then we must do so using both the terms like **unsigned char**.

Default values of Constants

Integer constants, by default, represent **int** type data. We can override this default by specifying unsigned or long after the number (by appending U or L) as shown below:

Literal	Type	Value
+111	int	111
-222	int	-222
45678U	unsigned int	45,678
-56789l	long int	-56,789
987654UL	unsigned long int	9,87,654

Similarly, floating point constants, by default represent **double** type data. If we want the resulting data type to be **float** or **long double**, we must append the letter f or F to the number for **float** and letter l or L for **long double** as shown below.

Literal	Type	Value
0.	double	0.0
.0	double	0.0
12.0	double	12.0
1.234	double	1.234
-1.2f	float	-1.2
1.23456789L	long double	1.23456789

User-Defined Type Declaration

C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables . It takes the general form:

```
typedef type identifier;
```

Where *type* refers to an existing data type and "identifier" refers to the "new" name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type is 'new' only in name, but not the data type. **typedef** cannot create a new type. Some examples of type definition are:

```
typedef int units;
typedef float marks;
```

Here, **units** symbolizes **int** and **marks** symbolizes **float.** They can be later used to declare variables as follows:

```
units batch1, batch2;
marks name1[50], name2[50];
```

batch1 and batch2 are inclared as **int** variable and name1[50] and name2[50] are declared as 50 element floating point array variables. The main advantage of **typedef** is that we can create meaningful data type names for increasing the readability of the program.

Another user-defined data type is enumerated data type provided by ANSI standard. It is defined as follows:

enum **identifier** {*value1, value2, ... valuen*};

The "identifier" is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as *enumeration constants*). After this definition, we can declare variables to be of this 'new' type as below:

```
enum identifier v1, v2, ... vn;
```

The enumerated variables v1, v2, ... vn can only have one of the values *value1, value2, ... valuen.* The assignments of the following types are valid:

```
v1 = value3;
v5 = value1;
```

An example:

```
enum day {Monday,Tuesday, ... Sunday};
enum day week_st, week_end;
week_st = Monday;
week_end = Friday;
if(week_st == Tuesday)
  week_end = Saturday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value1 is assigned 0, value2 is assigned 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants. For example:

```
enum day {Monday = 1, Tuesday, ... Sunday};
```

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement. Example:

```
enum day {Monday, ... Sunday} week_st, week_end;
```

2.9 DECLARATION OF STORAGE CLASS

Variables in C can have not only *data type* but also *storage class* that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognized. Consider the following example:

```
  /* Example of storage classes */
int  m;
main()
```

```
{
   int i;
   float balance;
   ....
   ....
   function1();
}
function1()
{
   int i;
   float sum;
   ....
   ....
}
```

The variable **m** which has been declared before the **main** is called *global* variable. It can be used in all the functions in the program. It need not be declared in other functions. A global variable is also known as an *external* variable.

The variables **i**, **balance** and **sum** are called *local* variables because they are declared inside a function. Local variables are visible and meaningful only inside the functions in which they are declared. They are not known to other functions. Note that the variable **i** has been declared in both the functions. Any change in the value of **i** in one function does not affect its value in the other.

C provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of variables. The concepts of scope and lifetime are important only in multifunction and multiple file programs and therefore the storage classes are considered in detail later when functions are discussed. For now, remember that there are four storage class specifiers (**auto, register, static**, and **extern**) whose meanings are given in Table 2.10.

The storage class is another qualifier (like **long** or **unsigned)** that can be added to a variable declaration as shown below:

```
auto int count;
register char ch;
static int x;
extern long total;
```

Static and external (**extern**) variables are automatically initialized to zero. Automatic (**auto**) variables contain undefined values (known as 'garbage') unless they are initialized explicitly.

Table 2.10 *Storage Classes and Their Meaning*

Storage class	*Meaning*
auto	Local variable known only to the function in which it is declared. *Default is auto.*
static	Local variable which exists and retains its value even after the control is transferred to the calling function.
extern	Global variable known to all functions in the file.
register	Local variable which is stored in the register.

2.10 ASSIGNING VALUES TO VARIABLES

Variables are created for use in program statements such as

```
value = amount + inrate * amount;
while (year <= PERIOD)
{
   ....
   ....
   year = year + 1;
}
```

In the first statement, the numeric value stored in the variable **inrate** is multiplied by the value stored in **amount** and the product is added to **amount**. The result is stored in the variable value. This process is possible only if the variables **amount** and inrate have already been given values. The variable **value** is called the *target variable*. While all the variables are declared for their type, the variables that are used in expressions (on the right side of equal (=) sign of a computational statement) *must* be assigned values before they are encountered in the program. Similarly, the variable **year** and the symbolic constant **PERIOD** in the **while** statement must be assigned values before this statement is encountered.

Assignment Statement

Values can be assigned to variables using the assignment operator = as follows:

```
variable_name = constant;
```

We have already used such statements in Chapter 1. Further examples are:

```
initial_value = 0;
final_value   = 100;
balance       = 75.84;
yes           = 'x';
```

C permits multiple assignments in one line. For example

```
initial_value = 0; final_value = 100;
```

are valid statements.

An assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity (or the expression) on the right. The statement

```
year = year + 1;
```

means that the 'new value' of **year** is equal to the 'old value' of **year** plus 1.

During assignment operation, C converts the type of value on the right-hand side to the type on the left. This may involve truncation when real value is converted to an integer.

It is also possible to assign a value to a variable at the time the variable is declared. This takes the following form:

Some examples are:

```
int final_value  =  100;
char yes         =  'x';
double balance   =  75.84;
```

The process of giving initial values to variables is called *initialization*. C permits the *initialization* of more than one variables in one statement using multiple assignment operators. For example the statements

```
p = q = s = 0;
x = y = z = MAX;
```

are valid. The first statement initializes the variables **p**, **q**, and **s** to zero while the second initializes **x**, **y**, and **z** with **MAX**. Note that **MAX** is a symbolic constant defined at the beginning.

Remember that external and static variables are initialized to zero by *default*. Automatic variables that are not initialized explicitly will contain garbage.

Example 2.2 Program in Fig. 2.8 shows typical declarations, assignments and values stored in various types of variables.

The variables **x** and **p** have been declared as floating-point variables. Note that the way the value of 1.234567890000 that we assigned to **x** is displayed under different output formats. The value of **x** is displayed as 1.234567880630 under %.12lf format, while the actual value assigned is 1.234567890000. This is because the variable **x** has been declared as a **float** that can store values only up to six decimal places.

The variable **m** that has been declared as **int** is not able to store the value 54321 correctly. Instead, it contains some garbage. Since this program was run on a 16-bit machine, the maximum value that an **int** variable can store is only 32767. However, the variable **k** (declared as **unsigned**) has stored the value 54321 correctly. Similarly, the **long int** variable **n** has stored the value 1234567890 correctly.

The value 9.87654321 assigned to **y** declared as double has been stored correctly but the value is printed as 9.876543 under %lf format. Note that unless specified otherwise, the **printf** function will always display a **float** or **double** value to six decimal places. We will discuss later the output formats for displaying numbers.

Program

```
main()
{
/*.........DECLARATIONS.............................*/
     float    x,p ;
     double   y,q ;
     unsigned k ;
/*.........DECLARATIONS AND ASSIGNMENTS.............*/
     int      m = 54321 ;
     long int n = 1234567890 ;
/*.........ASSIGNMENTS..............................*/
     x = 1.234567890000 ;
```

```
        y = 9.87654321 ;
        k = 54321 ;
        p = q = 1.0 ;
    /*.........PRINTING.............................*/
        printf("m = %d\n", m) ;
        printf("n = %ld\n", n) ;
        printf("x = %.12lf\n", x) ;
        printf("x = %f\n", x) ;
        printf("y = %.12lf\n",y) ;
        printf("y = %lf\n", y) ;
        printf("k = %u p = %f q = %.12lf\n", k, p, q) ;
    }

Output
        m = -11215
        n = 1234567890
        x = 1.234567880630
        x = 1.234568
        y = 9.876543210000
        y = 9.876543
        k = 54321 p = 1.000000 q = 1.000000000000
```

Fig. 2.8 *Examples of assignments*

Reading Data from Keyboard

Another way of giving values to variables is to input data through keyboard using the **scanf** function. It is a general input function available in C and is very similar in concept to the **printf** function. It works much like an INPUT statement in BASIC. The general format of **scanf** is as follows:

scanf("**control string**", **&variable1,&variable2,....**);

The control string contains the format of data being received. The ampersand symbol **&** before each variable name is an operator that specifies the variable name's *address*. We must always use this operator, otherwise unexpected results may occur. Let us look at an example:

```
scanf("%d", &number);
```

When this statement is encountered by the computer, the execution stops and waits for the value of the variable **number** to be typed in. Since the control string "%d" specifies that an integer value is to be read from the terminal, we have to type in the value in integer form. Once the number is typed in and the 'Return' Key is pressed, the computer then proceeds to the next statement. Thus, the use of **scanf** provides an interactive feature and makes the program 'user friendly'. The value is assigned to the variable **number**.

Example 2.3 The program in Fig. 2.9 illustrates the use of **scanf** function.

The first executable statement in the program is a **printf,** requesting the user to enter an integer number. This is known as "prompt message" and appears on the screen like

Enter an integer number

As soon as the user types in an integer number, the computer proceeds to compare the value with 100. If the value typed in is less than 100, then a message

Your number is smaller than 100

is printed on the screen. Otherwise, the message

Your number contains more than two digits

is printed. Outputs of the program run for two different inputs are also shown in Fig. 2.9.

```
Program
  main()
  {
       int number;

       printf("Enter an integer number\n");
       scanf ("%d", &number);

       if ( number < 100 )
         printf("Your number is smaller than 100\n\n");
       else
         printf("Your number contains more than two digits\n");
  }

Output
  Enter an integer number
  54
  Your number is smaller than 100
  Enter an integer number
  108
  Your number contains more than two digits
```

Fig. 2.9 *Use of scanf function for interactive computing*

Some compilers permit the use of the 'prompt message' as a part of the control string in **scanf**, like

```
scanf("Enter a number %d",&number);
```

We discuss more about **scanf** in Chapter 4.

In Fig. 2.9 we have used a decision statement **if...else** to decide whether the number is less than 100. Decision statements are discussed in depth in Chapter 5.

Example 2.4 Sample program 3 discussed in Chapter 1 can be converted into a more flexible interactive program using **scanf** as shown in Fig. 2.10.

In this case, computer requests the user to input the values of the amount to be invested, interest rate and period of investment by printing a prompt message

Input amount, interest rate, and period

and then waits for input values. As soon as we finish entering the three values corresponding to the

Program

```
main()
{
     int year, period ;
     float amount, inrate, value ;

     printf("Input amount, interest rate, and period\n\n") ;
     scanf ("%f %f %d", &amount, &inrate, &period) ;
     printf("\n") ;
     year = 1 ;

     while( year <= period )
     {
          value = amount + inrate * amount ;
          printf("%2d Rs %8.2f\n", year, value) ;
          amount = value ;
          year = year + 1 ;
     }
}
```

Output

```
Input amount, interest rate, and period

10000 0.14 5

  1 Rs 11400.00
  2 Rs 12996.00
  3 Rs 14815.44
  4 Rs 16889.60
  5 Rs 19254.15

Input amount, interest rate, and period

20000 0.12 7

  1 Rs 22400.00
  2 Rs 25088.00
  3 Rs 28098.56
  4 Rs 31470.39
  5 Rs 35246.84
  6 Rs 39476.46
  7 Rs 44213.63
```

Fig. 2.10 *Interactive investment program*

three variables **amount, inrate,** and **period,** the computer begins to calculate the amount at the end of each year, up to 'period' and produces output as shown in Fig. 2.10.

Note that the **scanf** function contains three variables. In such cases, care should be exercised to see that the values entered match the *order* and *type* of the variables in the list. Any mismatch might lead to unexpected results. The compiler may not detect such errors.

2.11 DEFINING SYMBOLIC CONSTANTS

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant **"pi".** Another example is the total number of students whose mark-sheets are analysed by a 'test analysis program'. The number of students, say 50, may be used for calculating the class total, class average, standard deviation, etc. We face two problems in the subsequent use of such programs.

1. Problem in modification of the program.
2. Problem in understanding the program.

Modifiability

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculations or the number 50 to 100 to process the test results of another class. In both the cases, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

Understandability

When a numeric value appears in a program, its use is not always clear, especially when the same value means different things in different places. For example, the number 50 may mean the number of students at one place and the 'pass marks' at another place of the same program. We may forget what a certain number meant, when we read the program some days later.

Assignment of such constants to a *symbolic name* frees us from these problems. For example, we may use the name **STRENGTH** to define the number of students and **PASS_MARK** to define the pass marks required in a subject. Constant values are assigned to these names at the beginning of the program. Subsequent use of the names **STRENGTH** and **PASS_MARK** in the program has the effect of causing their defined values to be automatically substituted at the appropriate points. A constant is defined as follows:

Defining symbolic names before use

Valid examples of constant definitions are:

```
#define STRENGTH 100
#define PASS_MARK 50
```

```
#define MAX 200
#define PI 3.14159
```

Symbolic names are sometimes called *constant identifiers*. Since the symbolic names are constants (not variables), they do not appear in declarations. The following rules apply to a **#define** statement which define a symbolic constant.

1. Symbolic names have the same form as variable names. (Symbolic names are written in CAPITALS to visually distinguish them from the normal variable names, which are written in lowercase letters. This is only a convention, not a rule.)
2. No blank space between the pound sign '#' and the word **define** is permitted.
3. '#' must be the first character in the line.
4. A blank space is required between **#define** and *symbolic name* and between the ***symbolic name*** and the *constant*.
5. **#define** statements must not end with a semicolon.
6. After definition, the *symbolic name* should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH = 200; is illegal.
7. Symbolic names are NOT declared for data types. Its data type depends on the type of constant.
8. **#define** statements may appear *anywhere* in the program but before it is referenced in the program (the usual practice is to place them in the beginning of the program).

#define statement is a *preprocessor* compiler directive and is much more powerful than what has been mentioned here. More advanced types of definitions will be discussed later. Table 2.11 illustrates some invalid statements of **#define.**

Table 2.11 *Examples of Invalid #define Statements*

Statement	*Validity*	*Remark*
#define X = 2.5	Invalid	'=' sign is not allowed
# define MAX 10	Invalid	No white space between # and define
#define N 25;	Invalid	No semicolon at the end
#define N 5, M 10	Invalid	A statement can define only one name.
#Define ARRAY 11	Invalid	define should be in lowercase letters
#define PRICE$ 100	Invalid	$ symbol is not permitted in name

2.12 DECLARING A VARIABLE AS CONSTANT

We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier **const** at the time of initialization. Example:

```
const int class_size = 40;
```

const is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the **int** variable **class_size** must not be modified by the program. However, it can be used on the right_hand side of an assignment statement like any other variable.

2.13 DECLARING A VARIABLE AS VOLATILE

ANSI standard defines another qualifier **volatile** that could be used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources (from outside the program). For example:

```
volatile int date;
```

The value of **date** may be altered by some external factors even if it does not appear on the left-hand side of an assignment statement. When we declare a variable as **volatile**, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

Remember that the value of a variable declared as **volatile** can be modified by its own program as well. If we wish that the value must not be modified by the program while it may be altered by some other process, then we may declare the variable as both **const** and **volatile** as shown below:

```
volatile const int location = 100;
```

2.14 OVERFLOW AND UNDERFLOW OF DATA

Problem of data overflow occurs when the value of a variable is either too big or too small for the data type to hold. The largest value that a variable can hold also depends on the machine. Since floating-point values are rounded off to the number of significant digits allowed (or specified), an overflow normally results in the largest possible real value, whereas an underflow results in zero.

Integers are always exact within the limits of the range of the integral data types used. However, an overflow which is a serious problem may occur if the data type does not match the value of the constant. C does not provide any warning or indication of integer overflow. It simply gives incorrect results. (Overflow normally produces a negative number.) We should therefore exercise a greater care to define correct data types for handling the input/output values.

Just Remember

- Do not use the underscore as the first character of identifiers (or variable names) because many of the identifiers in the system library start with underscore.
- Use only 31 or less characters for identifiers. This helps ensure portability of programs.
- Do not use keywords or any system library names for identifiers.
- Use meaningful and intelligent variable names.
- Do not create variable names that differ only by one or two letters.
- Each variable used must be declared for its type at the beginning of the program or function.
- All variables must be initialized before they are used in the program.
- Integer constants, by default, assume **int** types. To make the numbers **long** or **unsigned**, we must append the letters L and U to them.
- Floating point constants default to **double**. To make them to denote **float** or **long double**, we must append the letters F or L to the numbers.

- Do not use lowercase l for long as it is usually confused with the number 1.
- Use single quote for character constants and double quotes for string constants.
- A character is stored as an integer. It is therefore possible to perform arithmetic operations on characters.
- Do not combine declarations with executable statements.
- A variable can be made constant either by using the preprocessor command **#define** at the beginning of the program or by declaring it with the qualifier **const** at the time of initialization.
- Do not use semicolon at the end of **#define** directive.
- The character # should be in the first column.
- Do not give any space between # and **define.**
- C does not provide any warning or indication of overflow. It simply gives incorrect results. Care should be exercised in defining correct data type.
- A variable defined before the main function is available to all the functions in the program.
- A variable defined inside a function is local to that function and not available to other functions.

CASE STUDIES

1. Calculation of Average of Numbers

A program to calculate the average of a set of N numbers is given in Fig. 2.11.

```
Program
  #define    N    10              /* SYMBOLIC CONSTANT */
  main()
  {
       int    count ;                /* DECLARATION OF */
       float  sum, average, number ; /* VARIABLES */
       sum    = 0 ;                  /* INITIALIZATION */
       count  = 0 ;                  /* OF VARIABLES */
       while( count < N )
       {
              scanf("%f", &number) ;
              sum = sum + number ;
              count = count + 1 ;
       }
       average = sum/N ;
       printf("N = %d Sum = %f", N, sum);
       printf(" Average = %f", average);
  }
```

```
Output
   1
   2.3
   4.67
   1.42
   7
   3.67
   4.08
   2.2
   4.25
   8.21
   N = 10    Sum = 38.799999 Average = 3.880000
```

Fig. 2.11 *Average of N numbers*

The variable **number** is declared as **float** and therefore it can take both integer and real numbers. Since the symbolic constant **N** is assigned the value of 10 using the **#define** statement, the program accepts ten values and calculates their sum using the **while** loop. The variable **count** counts the number of values and as soon as it becomes 11, the **while** loop is exited and then the average is calculated.

Notice that the actual value of sum is 38.8 but the value displayed is 38.799999. In fact, the actual value that is displayed is quite dependent on the computer system. Such an inaccuracy is due to the way the floating point numbers are internally represented inside the computer.

Temperature Conversion Problem

The program presented in Fig. 2.12 converts the given temperature in fahrenheit to celsius using the following conversion formula:

$$C = \frac{F - 32}{1.8}$$

```
Program
   #define F_LOW     0       /* ---------------- */
   #define F_MAX   250       /* SYMBOLIC CONSTANTS */
   #define STEP     25       /* ---------------- */

   main()
   {
        typedef float REAL ;          /* TYPE DEFINITION */
        REAL fahrenheit, celsius ;    /* DECLARATION */

        fahrenheit = F_LOW ;          /* INITIALIZATION */
        printf("Fahrenheit Celsius\n\n") ;
        while( fahrenheit <= F_MAX )
        {
             celsius = ( fahrenheit - 32.0 ) / 1.8 ;
```

```
            printf(" %5.1f %7.2f\n", fahrenheit, celsius);
            fahrenheit = fahrenheit + STEP ;
        }
    }
```

Output

```
Fahrenheit      Celsius

     0.0          -17.78
    25.0           -3.89
    50.0           10.00
    75.0           23.89
   100.0           37.78
   125.0           51.67
   150.0           65.56
   175.0           79.44
   200.0           93.33
   225.0          107.22
   250.0          121.11
```

Fig. 2.12 *Temperature conversion—fahrenheit-celsius*

The program prints a conversion table for reading temperature in celsius, given the fahrenheit values. The minimum and maximum values and step size are defined as symbolic constants. These values can be changed by redefining the **#define** statements. An user-defined data type name **REAL** is used to declare the variables **fahrenheit** and **celsius**.

The formation specifications %5.1f and %7.2 in the second **printf** statement produces two-column output as shown.

REVIEW QUESTIONS

2.1 State whether the following statements are *true* or *false*.

(a) Any valid printable ASCII character can be used in an identifier.
(b) All variables must be given a type when they are declared.
(c) Declarations can appear anywhere in a program.
(d) ANSI C treats the variables **name** and **Name** to be same.
(e) The underscore can be used anywhere in an identifier.
(f) The keyword **void** is a data type in C.
(g) Floating point constants, by default, denote **float** type values.
(h) Like variables, constants have a type.
(i) Character constants are coded using double quotes.
(j) Initialization is the process of assigning a value to a variable at the time of declaration.
(k) All **static** variables are automatically initialized to zero.
(l) The **scanf** function can be used to read only one value at a time.

2.2 Fill in the blanks with appropriate words.

(a) The keyword _____ can be used to create a data type identifier.
(b) _____ is the largest value that an unsigned short int type variable can store.

(c) A global variable is also known as _____ variable.
(d) A variable can be made constant by declaring it with the qualifier _____ at the time of initialization.

2.3 What are trigraph characters? How are they useful?
2.4 Describe the four basic data types. How could we extend the range of values they represent?
2.5 What is an unsigned integer constant? What is the significance of declaring a constant unsigned?
2.6 Describe the characteristics and purpose of escape sequence characters.
2.7 What is a variable and what is meant by the "value" of a variable?
2.8 How do variables and symbolic names differ?
2.9 State the differences between the declaration of a variable and the definition of a symbolic name.
2.10 What is initialization? Why is it important?
2.11 What are the qualifiers that an **int** can have at a time?
2.12 A programmer would like to use the word DPR to declare all the double-precision floating point values in his program. How could he achieve this?
2.13 What are enumeration variables? How are they declared? What is the advantage of using them in a program?
2.14 Describe the purpose of the qualifiers **const** and **volatile**.
2.15 When dealing with very small or very large numbers, what steps would you take to improve the accuracy of the calculations?
2.16 Which of the following are invalid constants and why?

0.0001	5x1.5	99999
+100	75.45 E-2	"15.75"
–45.6	–1.79 e + 4	0.00001234

2.17 Which of the following are invalid variable names and why?

Minimum	First.name	n1+n2	&name
doubles	3rd_row	n$	Row1
float	Sum Total	Row Total	Column-total

2.18 Find errors, if any, in the following declaration statements.

```
Int x;
float letter,DIGIT;
double = p,q
exponent alpha,beta;
m,n,z: INTEGER
short char c;
long int m; count;
long float temp;
```

2.19 What would be the value of x after execution of the following statements?

```
int x, y = 10;
char z = 'a';
x = y + z;
```

2.20 Identify syntax errors in the following program. After corrections, what output would you expect when you execute it?

```
#define PI 3.14159
main()
{
     int R,C;           /* R-Radius of circle
     float perimeter;   /* Circumference of circle */
     float area;        /* Area of circle */
     C = PI
     R = 5;
     Perimeter = 2.0 * C *R;
     Area   = C*R*R;
     printf("%f", "%d",&perimeter,&area)
     }
```

PROGRAMMING EXERCISES

2.1 Write a program to determine and print the sum of the following harmonic series for a given value of n:

$$1 + 1/2 + 1/3 + ... + 1/n$$

The value of n should be given interactively through the terminal.

2.2 Write a program to read the price of an item in decimal form (like 15.95) and print the output in paise (like 1595 paise).

2.3 Write a program that prints the even numbers from 1 to 100.

2.4 Write a program that requests two float type numbers from the user and then divides the first number by the second and display the result along with the numbers.

2.5 The price of one kg of rice is Rs. 16.75 and one kg of sugar is Rs. 15. Write a program to get these values from the user and display the prices as follows:

*** LIST OF ITEMS ***

Item	Price
Rice	Rs 16.75
Sugar	Rs 15.00