4

Managing Input and Output Operations

4.1 INTRODUCTION

Reading, processing, and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data, often known as *information* or *results*, on a suitable medium. So far we have seen two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements such as **x = 5; a = 0;** and so on. Another method is to use the input function **scanf** which can read data from a keyboard. We have used both the methods in most of our earlier example programs. For outputting results we have used extensively the function **printf** which sends results out to a terminal.

Unlike other high-level languages, C does not have any built-in input/output statements as part of its syntax. All input/output operations are carried out through function calls such as **printf** and **scanf**. There exist several functions that have more or less become standard for input and output operations in C. These functions are collectively known as the standard I/O library. In this chapter we shall discuss some common I/O functions that can be used on many machines without any change. However, one should consult the system reference manual for exact details of these functions and also to see what other functions are available.

It may be recalled that we have included a statement

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

in the Sample Program 5 in Chapter 1, where a math library function cos(x) has been used. This is to instruct the compiler to fetch the function cos(x) from the math library, and that it is not a part of C language. Similarly, each program that uses a standard input/output function must contain the statement

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

at the beginning. However, there might be exceptions. For example, this is not necessary for the functions **printf** and **scanf** which have been defined as a part of the C language.

The file name **stdio.h** is an abbreviation for *standard input-output header* file. The instruction **#include** *<stdio.h>* tells the compiler to search for a file named **stdio.h** and place its contents at this point in the program. The contents of the header file become part of the source code when it is compiled.

4.2 READING A CHARACTER

The simplest of all input/output operations is reading a character from the 'standard input' unit (usually the keyboard) and writing it to the 'standard output' unit (usually the screen). Reading a single character can be done by using the function **getchar.** (This can also be done with the help of the **scanf** function which is discussed in Section 4.4.) The **getchar** takes the following form:

***variable_name* = getchar( );**

variable_name is a valid C name that has been declared as **char** type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to **getchar** function. Since **getchar** is used on the right-hand side of an assignment statement, the character value of **getchar** is in turn assigned to the variable name on the left. For example

```
char name;
name = getchar();
```

Will assign the character 'H' to the variable **name** when we press the key H on the keyboard. Since **getchar** is a function, it requires a set of parentheses as shown.

Example 4.1 The program in Fig. 4.1 shows the use of **getchar** function in an interactive environment.

The program displays a question of YES/NO type to the user and reads the user's response in a single character (Y or N). If the response is Y or y, it outputs the message

My name is BUSY BEE

otherwise, outputs

You are good for nothing

NOTE: There is one line space between the input text and output message.

Program

```
#include <stdio.h>
main()
{
   char answer;
   printf("Would you like to know my name?\n");

   printf("Type Y for YES and N for NO: ");
   answer = getchar(); /* .... Reading a character...*/
```

```
        if(answer == 'Y' || answer == 'y')
          printf("\n\nMy name is BUSY BEE\n");
        else
          printf("\n\nYou are good for nothing\n");
    }
Output
    Would you like to know my name?
    Type Y for YES and N for NO: Y

    My name is BUSY BEE

    Would you like to know my name?
    Type Y for YES and N for NO: n

    You are good for nothing
```

Fig. 4.1 *Use of* **getchar** *function to read a* **character** *from keyboard*

The **getchar** function may be called successively to read the characters contained in a ine of text. For example, the following program segment reads characters from keyboard one after another until the 'Return' key is pressed.

```
--------

--------
char character;
character = ' ';
while(character != '\n')
{
    character = getchar();
}
--------

--------
```

WARNING

The **getchar()** function accepts any character keyed in. This includes RETURN and TAB. This means when we enter single character input, the newline character is waiting in the input queue after **getchar()** returns. This could create problems when we use **getchar()** in a loop interactively. A dummy **getchar()** may be used to 'eat' the unwanted newline character. We can also use the **fflush** function to flush out the unwanted characters.

NOTE: We shall be using decision statements like **if**, **if...else** and **while** extensively in this chapter. They are discussed in detail in Chapters 5 and 6.

Example 4.2 The program of Fig. 4.2 requests the user to enter a character and displays a message on the screen telling the user whether the character is an alphabet or digit, or any other special character.

This program receives a character from the keyboard and tests whether it is a letter or digit and prints out a message accordingly. These tests are done with the help of the following functions:

```
isalpha(character)
isdigit(character)
```

For example, **isalpha** assumes a value non-zero (TRUE) if the argument **character** contains an alphabet; otherwise it assumes 0 (FALSE). Similar is the case with the function **isdigit**.

Program:

```
#include <stdio.h>
#include <ctype.h>
main()
{
  char character;
  printf("Press any key\n");
  character = getchar();
  if (isalpha(character) > 0)/* Test for letter */
    printf("The character is a letter.");
  else
    if (isdigit (character) > 0)/* Test for digit */
      printf("The character is a digit.");
    else
      printf("The character is not alphanumeric.");
}
```

Output

```
Press any key
h
The character is a letter.
Press any key
5
The character is a digit.
Press any key
*
The character is not alphanumeric.
```

Fig. 4.2 *Program to test the* ***character*** *type*

C supports many other similar functions, which are given in Table 4.1. These character functions are contained in the file **ctype.h** and therefore the statement

```
#include <ctype.h>
```

must be included in the program.

Table 4.1 *Character Test Functions*

Function	*Test*
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c lower case letter?
isprint(c)	Is c a printable character?
ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a white space character?
isupper(c)	Is c an upper case letter?

4.3 WRITING A CHARACTER

Like **getchar**, there is an analogous function **putchar** for writing characters one at a time to the terminal. It takes the form as shown below:

putchar *(variable_name);*

where ***variable_name*** is a type **char** variable containing a character. This statement displays the character contained in the ***variable_name*** at the terminal. For example, the statements

```
answer = 'Y';
putchar (answer);
```

will display the character Y on the screen. The statement

```
putchar ('\n');
```

would cause the cursor on the screen to move to the beginning of the next line.

Example 4.3 A program that reads a character from keyboard and then prints it in reverse case is given in Fig. 4.3. That is, if the input is upper case, the output will be lower case and vice versa.

The program uses three new functions: **islower, toupper**, and **tolower**. The function **islower** is a conditional function and takes the value TRUE if the argument is a lowercase alphabet; otherwise takes the value FALSE. The function **toupper** converts the lowercase argument into an uppercase alphabet while the function **tolower** does the reverse.

```
Program
    #include <stdio.h>
    #include <ctype.h>
    main()
    {
      char alphabet;
      printf("Enter an alphabet");
      putchar('\n'); /* move to next line */
      alphabet = getchar();
      if (islower(alphabet))
```

```
            putchar(toupper(alphabet));/* Reverse and display */
        else
            putchar(tolower(alphabet)); /* Reverse and display */
    }
Output

    Enter an alphabet
    a
    A
    Enter an alphabet
    Q
    q
    Enter an alphabet
    z
    Z
```

Fig. 4.3 *Reading and writing of alphabets in reverse case*

4.4 FORMATTED INPUT

Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data:

```
15.75 123 John
```

This line contains three pieces of data, arranged in a particular form. Such data has to be read conforming to the format of its appearance. For example, the first part of the data should be read into a variable **float**, the second into **int**, and the third part into **char**. This is possible in C using the **scanf** function. (**scanf** means *scan* formatted.)

We have already used this input function in a number of examples. Here, we shall explore all of the options that are available for reading the formatted data with **scanf** function. The general form of **scanf** is

scanf (*"control string", arg1, arg2, ... argn);*

The *control string* specifies the field format in which the data is to be entered and the arguments *arg*1, *arg2*, ..., *argn* specify the address of locations where the data is stored. Control string and arguments are separated by commas.

Control string (also known as *format string*) contains field specifications, which direct the interpretation of input data. It may include:

- Field (or format) specifications, consisting of the conversion character %, a data type character (or type specifier), and an *optional* number, specifying the field width.
- Blanks, tabs, or newlines.

Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional. The discussions that follow will clarify these concepts.

Inputting Integer Numbers

The field specification for reading an integer number is:

```
% w sd
```

The percentage sign (%) indicates that a conversion specification follows. ***w*** is an integer number that specifies the *field width* of the number to be read and **d**, known as data type character, indicates that the number to be read is in integer mode. Consider the following example:

```
scanf ("%2d %5d", &num1, &num2);
```

Data line:

```
50  31426
```

The value 50 is assigned to **num1** and 31426 to **num2**. Suppose the input data is as follows:

```
31426 50
```

The variable **num1** will be assigned 31 (because of %2d) and **num2** will be assigned 426 (unread part of 31426). The value 50 that is unread will be assigned to the first variable in the next **scanf** call. This kind of errors may be eliminated if we use the field specifications without the field width specifications. That is, the statement

```
scanf("%d %d", &num1, &num2);
```

will read the data

```
31426 50
```

correctly and assign 31426 to **num1** and 50 to **num2.**

Input data items must be separated by spaces, tabs or newlines. Punctuation marks do not count as separators. When the **scanf** function searches the input data line for a value to be read, it will always bypass any white space characters.

What happens if we enter a floating point number instead of an integer? The fractional part may be stripped away! Also, **scanf** may skip reading further input.

When the **scanf** reads a particular value, reading of the value will be terminated as soon as the number of characters specified by the field width is reached (if specified) or until a character that is not valid for the value being read is encountered. In the case of integers, valid characters are an optionally signed sequence of digits.

An input field may be skipped by specifying * in the place of field width. For example, the statement

```
scanf("%d %*d %d", &a, &b)
```

will assign the data

```
123 456 789
```

as follows:

```
123 to a
456 skipped (because of *)
789 to b
```

The data type character **d** may be preceded by 'l' (letter ell) to read long integers and **h** to read short integers.

NOTE: We have provided white space between the field specifications. These spaces are not necessary with the numeric input, but it is a good practice to include them.

Example 4.4 Various input formatting options for reading integers are experimented in the program shown in Fig. 4.4.

```
Program
    main()
    {
       int a,b,c,x,y,z;
       int p,q,r;
       printf("Enter three integer numbers\n");
       scanf("%d %*d %d",&a,&b,&c);
       printf("%d %d %d \n\n",a,b,c);
       printf("Enter two 4-digit numbers\n");
       scanf("%2d %4d",&x,&y);
       printf("%d %d\n\n", x,y);
       printf("Enter two integers\n");
       scanf("%d %d", &a,&x);
       printf("%d %d \n\n",a,x);
       printf("Enter a nine digit number\n");
       scanf("%3d %4d %3d",&p,&q,&r);
       printf("%d %d %d \n\n",p,q,r);
       printf("Enter two three digit numbers\n");
       scanf("%d %d",&x,&y);
       printf("%d %d",x,y);
    }
Output
       Enter three integer numbers
       1 2 3
       1 3 -3577
       Enter two 4-digit numbers
       6789 4321
       67 89
       Enter two integers
       44 66
       4321 44
    Enter a nine-digit number
    123456789
    66 1234 567
    Enter two three-digit numbers
    123 456
    89 123
```

Fig. 4.4 *Reading integers using* ***scanf***

The first **scanf** requests input data for three integer values **a, b,** and **c,** and accordingly three values 1, 2, and 3 are keyed in. Because of the specification %*d the value 2 has been skipped and 3 is assigned to the variable **b.** Notice that since no data is available for **c,** it contains garbage.

The second **scanf** specifies the format %2d and %4d for the variables **x** and **y** respectively. Whenever we specify field width for reading integer numbers, the input numbers should not contain more digits that the specified size. Otherwise, the extra digits on the right-hand side will be truncated and assigned to the next variable in the list. Thus, the second **scanf** has truncated the four digit number 6789 and assigned 67 to **x** and 89 to **y**. The value 4321 has been assigned to the first variable in the immediately following **scanf** statement.

NOTE: It is legal to use a non-whitespace character between field specifications. However, the **scanf** expects a matching character in the given location. For example,

```
scanf("%d-%d", &a, &b);
```

accepts input like

```
123-456
```

to assign 123 to a and 456 to b.

Inputting Real Numbers

Unlike integer numbers, the field width of real numbers is not to be specified and therefore **scanf** reads real numbers using the simple specification **%f** for both the notations, namely decimal point notation and exponential notation. For example, the statement

```
scanf("%f %f %f", &x, &y, &z);
```

with the input data

```
475.89 43.21E-1 678
```

will assign the value 475.89 to **x**, 4.321 to **y**, and 678.0 to **z**. The input field specifications may be separated by any arbitrary blank spaces.

If the number to be read is of **double** type, then the specification should be **%lf** instead of simple **%f**. A number may be skipped using **%*f** specification.

Example 4.5 Reading of real numbers (in both decimal point and exponential notation) is illustrated in Fig. 4.5.

```
Program
    main()
    {
      float x,y;
      double p,q;
      printf("Values of x and y:");
      scanf("%f %e", &x, &y);
      printf("\n");
      printf("x = %f\ny = %f\n\n", x, y);
      printf("Values of p and q:");
```

```
        scanf("%lf %lf", &p, &q);
        printf("\n\np = %.12lf\np = %.12e", p,q);
    }
```

Output

```
Values of x and y:12.3456 17.5e-2
x = 12.345600
y = 0.175000

Values of p and q:4.142857142857 18.5678901234567890

p = 4.142857142857
q = 1.856789012346e+001
```

Fig. 4.5 *Reading of* ***real*** *numbers*

Inputting Character Strings

We have already seen how a single character can be read from the terminal using the **getchar** function. The same can be achieved using the **scanf** function also. In addition, a **scanf** function can input strings containing more than one character. Following are the specifications for reading character strings:

%ws or %wc

The corresponding argument should be a pointer to a character array. However, %c may be used to read a single character when the argument is a pointer to a **char** variable.

Example 4.6 Reading of strings using **%wc** and **%ws** is illustrated in Fig. 4.6.

The program in Fig. 4.6 illustrates the use of various field specifications for reading strings. When we use **%wc** for reading a string, the system will wait until the w^{th} character is keyed in. **Note** that the specification **%s** terminates reading at the encounter of a blank space. Therefore, **name2** has read only the first part of "New York" and the second part is automatically assigned to **name3**. However, during the second run, the string "New-York" is correctly assigned to **name2**.

Program

```
main()
{
   int no;
   char name1[15], name2[15], name3[15];
   printf("Enter serial number and name one\n");
   scanf("%d %15c", &no, name1);
   printf("%d %15s\n\n", no, name1);
   printf("Enter serial number and name two\n");
```

```
        scanf("%d %s", &no, name2);
        printf("%d %15s\n\n", no, name2);
        printf("Enter serial number and name three\n");
        scanf("%d %15s", &no, name3);
        printf("%d %15s\n\n", no, name3);
    }
```

Output

```
Enter serial number and name one
1 123456789012345
1 123456789012345r
Enter serial number and name two
2 New York
2              New
Enter serial number and name three
2              York
Enter serial number and name one
1 123456789012
1 123456789012r
Enter serial number and name two
2 New-York
2          New-York
Enter serial number and name three
3 London
3          London
```

Fig. 4.6 *Reading of strings*

Some versions of **scanf** support the following conversion specifications for strings:

```
%[characters]
%[^characters]
```

The specification **%[characters]** means that only the characters specified within th brackets are permissible in the input string. If the input string contains any other characte the string will be terminated at the first encounter of such a character. The specificatio **%[^characters]** does exactly the reverse. That is, the characters specified after th circumflex (^) are not permitted in the input string. The reading of the string will b terminated at the encounter of one of these characters.

Example 4.7 The program in Fig. 4.7 illustrates the function of %( ) specification.

Program-A

```
main()
{
  char address[80];
```

```
        printf("Enter address\n");
        scanf("%[a-z]", address);
        printf("%-80s\n\n", address);
    }
Output
        Enter address
        new delhi 110002
        new delhi

Program-B
    main()
    {
        char address[80];
        printf("Enter address\n");
        scanf("%[^\n]", address);
        printf("%-80s", address);
    }
Output
        Enter address
        New Delhi 110 002
        New Delhi 110 002
```

Fig. 4.7 *Illustration of conversion specification%[] for strings*

Reading Blank Spaces

We have earlier seen that %s specifier cannot be used to read strings with blank spaces. But, this can be done with the help of %[ ] specification. Blank spaces may be included within the brackets, thus enabling the **scanf** to read strings with spaces. Remember that the lowercase and uppercase letters are distinct. See Fig. 4.7.

Reading Mixed Data Types

It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, care should be exercised to ensure that the input data items match the control specifications *in order* and *type*. When an attempt is made to read an item that does not match the type expected, the **scanf** function does not read any further and immediately returns the values read. The statement

```
scanf ("%d %c %f %s", &count, &code, &ratio, name);
```

will read the data

```
15 p 1.575 coffee
```

correctly and assign the values to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice versa, and the input data is converted to the type specified in the control string.

NOTE: A space before the *%c* specification in the format string is necessary to skip the white space before p.

Detection of Errors in Input

When a **scanf** function completes reading its list, it returns the value of number of items that are successfully read. This value can be used to test whether any errors occurred in reading the input. For example, the statement

```
scanf("%d %f %s, &a, &b, name);
```

will return the value 3 if the following data is typed in:

```
20 150.25 motor
```

and will return the value 1 if the following line is entered

```
20 motor 150.25
```

This is because the function would encounter a string when it was expecting a floating-point value, and would therefore terminate its scan after reading the first value.

Example 4.8 The program presented in Fig.4.8 illustrates the testing for correctness of reading of data by **scanf** function.

The function **scanf** is expected to read three items of data and therefore, when the values for all the three variables are read correctly, the program prints out their values. During the third run, the second item does not match with the type of variable and therefore the reading is terminated and the error message is printed. Same is the case with the fourth run.

In the last run, although data items do not match the variables, no error message has been printed. When we attempt to read a real number for an **int** variable, the integer part is assigned to the variable, and the truncated decimal part is assigned to the next variable.

NOTE: The character '2' is assigned to the character variable c.

```
Program
    main()
    {
      int a;
      float b;
      char c;
      printf("Enter values of a, b and c\n");
      if (scanf("%d %f %c", &a, &b, &c) == 3)
        printf("a = %d b = %f c = %c\n" , a, b, c);
      else
        printf("Error in input.\n");
    }
```

```
Output
        Enter values of a, b and c
            12 3.45 A
            a = 12    b = 3.450000     c = A
            Enter values of a, b and c
            23 78 9
            a = 23    b = 78.000000    c = 9
            Enter values of a, b and c
            8 A 5.25
            Error in input.
            Enter values of a, b and c
            Y 12 67
            Error in input.
            Enter values of a, b and c
            15.75 23 X
            a = 15    b = 0.750000     c = 2
```

Fig. 4.8 *Detection of errors in* ***scanf*** *input*

Commonly used **scanf** format codes are given in Table 4.2

Table 4.2 *Commonly used scanf Format Codes*

Code	*Meaning*
%c	read a single character
%d	read a decimal integer
%e	read a floating point value
%f	read a floating point value
%g	read a floating point value
%h	read a short integer
%i	read a decimal, hexadecimal or octal integer
%o	read an octal integer
%s	read a string
%u	read an unsigned decimal integer
%x	read a hexadecimal integer
%[..]	read a string of word(s)

The following letters may be used as prefix for certain conversion characters.

- h for short integers
- l for long integers or double
- L for long double

NOTE: C99 adds some more format codes. See the Appendix "C99 Features".

Points to Remember While Using scanf

If we do not plan carefully, some 'crazy' things can happen with **scanf**. Since the I/O routines are not a part of C language, they are made available either as a separate module of the C

library or as a part of the operating system (like UNIX). New features are added to these routines from time to time as new versions of systems are released. We should consult the system reference manual before using these routines. Given below are some of the general points to keep in mind while writing a **scanf** statement.

1. All function arguments, except the control string, *must* be pointers to variables.
2. Format specifications contained in the control string should match the arguments in order.
3. Input data items must be separated by spaces and must match the variables receiving the input in the same order.
4. The reading will be terminated, when **scanf** encounters a 'mismatch' of data or a character that is not valid for the value being read.
5. When searching for a value, **scanf** ignores line boundaries and simply looks for the next appropriate character.
6. Any unread data items in a line will be considered as part of the data input line to the next **scanf** call.
7. When the field width specifier *w* is used, it should be large enough to contain the input data size.

Rules for scanf

- Each variable to be read must have a filed specification.
- For each field specification, there must be a variable address of proper type.
- Any non-whitespace character used in the format string must have a match ing character in the user input.
- Never end the format string with whitespace. It is a fatal error!
- The scanf reads until:
  - A whitespace character is found in a numberic specification, or
  - The maximum number of characters have been read or
  - An error is detected, or
  - The end of file is reached

4.5 FORMATTED OUTPUT

We have seen the use of **printf** function for printing captions and numerical results. It is highly desirable that the outputs are produced in such a way that they are understandable and are in an easy-to-use form. It is therefore necessary for the programmer to give careful consideration to the appearance and clarity of the output produced by his program.

The **printf** statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals. The general form of **printf** statement is:

```
printf("control string", arg1, arg2, ....., argn);
```

Control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. *Escape sequence* characters such as \n, \t, and \b.

The control string indicates how many arguments follow and what their types are. The arguments *arg1, arg2, ..., argn* are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

A simple format specification has the following form:

```
% w.p type-specifier
```

where *w* is an integer number that specifies the total number of columns for the output value and *p* is another integer number that specifies the number of digits to the right of the decimal point (of a real number) or the number of characters to be printed from a string. Both *w* and *p* are optional. Some examples of formatted **printf** statement are:

```
printf("Programming in C");
printf(" ");
printf("\n");
printf("%d", x);
printf("a = %f\n b = %f", a, b);
printf("sum = %d", 1234);
printf("\n\n");
```

printf never supplies a *newline* automatically and therefore multiple **printf** statements may be used to build one line of output. A *newline* can be introduced by the help of a newline character '\n' as shown in some of the examples above.

Output of Integer Numbers

The format specification for printing an integer number is:

```
% w d
```

where *w* specifies the minimum field width for the output. However, if a number is greater than the specified field width, it will be printed in full, overriding the minimum specification. *d* specifies that the value to be printed is an integer. The number is written *right-justified* in the given field width. Leading blanks will appear as necessary. The following examples illustrate the output of the number 9876 under different formats:

Format	*Output*
printf("%d", 9876)	9 8 7 6
printf("%6d", 9876)	␣ ␣ 9 8 7 6
printf("%2d", 9876)	9 8 7 6
printf("%-6d", 9876)	9 8 7 6 ␣ ␣
printf("%06d", 9876)	0 0 9 8 7 6

It is possible to force the printing to be left-*justified* by placing a *minus* sign directly after the % character, as shown in the fourth example above. It is also possible to pad with zeros the leading blanks by placing a 0 (zero) before the field width specifier as shown in the last item above. The minus (–) and zero (0) are known as *flags*.

Long integers may be printed by specifying **ld** in the place of **d** in the format specifications. Similarly, we may use **hd** for printing short integers.

Example 4.9 The program in Fig. 4.9 illustrates the output of integer numbers under various formats.

```
Program
    main()
    {
      int m = 12345;
      long n = 987654;
      printf("%d\n",m);
      printf("%10d\n",m);
      printf("%010d\n",m);
      printf("%-10d\n",m);
      printf("%10ld\n",n);
      printf("%10ld\n",-n);
    }
Output
      12345
           12345
      0000012345
      12345
           987654
          -987654
```

Fig. 4.9 *Formatted output of integers*

Output of Real Numbers

The output of a real number may be displayed in decimal notation using the following format specification:

% w.p f

The integer *w* indicates the minimum number of positions that are to be used for the display of the value and the integer *p* indicates the number of digits to be displayed after the decimal point (*precision*). The value, when displayed, is *rounded to p decimal places* and printed *right-justified* in the field of *w* columns. Leading blanks and trailing zeros will appear as necessary. The default precision is 6 decimal places. The negative numbers will be printed with the minus sign. The number will be displayed in the form [–] mmm-nnn.

We can also display a real number in exponential notation by using the specification:

% w.p e

The display takes the form

[-] m.nnnne[±]xx

where the length of the string of n's is specified by the precision *p*. The default precision is 6. The field width **w** should satisfy the condition.

$$w \geq p+7$$

The value will be rounded off and printed right justified in the field of **w** columns.

Padding the leading blanks with zeros and printing with *left-justification* are also possible by using flags 0 or – before the field width specifier **w**.

The following examples illustrate the output of the number y = 98.7654 under different format specifications:

Format	*Output*
printf("%7.4f",y)	9 8 . 7 6 5 4
printf("%7.2f",y)	9 8 . 7 7
printf("%-7.2f",y)	9 8 . 7 7
printf("%f",y)	9 8 . 7 6 5 4
printf("%10.2e",y)	9 . 8 8 e + 0 1
printf("%11.4e",-y)	– 9 . 8 7 6 5 e + 0 1
printf("%-10.2e",y)	9 . 8 8 e + 0 1
printf("%e",y)	9 . 8 7 6 5 4 0 e + 0 1

Some systems also support a special field specification character that lets the user define the field size at run time. This takes the following form:

printf("%*.*f", *width, precision, number);

In this case, both the field width and the precision are given as arguments which will supply the values for **w** and **p**. For example,

```
printf("%*.*f",7,2,number);
```

is equivalent to

```
printf("%7.2f",number);
```

The advantage of this format is that the values for *width* and *precision* may be supplied at run time, thus making the format a *dynamic* one. For example, the above statement can be used as follows:

```
int width = 7;
int precision = 2;
.........
.........
printf("%*.*f", width, precision, number);
```

Example 4.10 All the options of printing a real number are illustrated in Fig. 4.10.

```
Program
    main()
    {
      float y = 98.7654;
      printf("%7.4f\n", y);
      printf("%f\n", y);
      printf("%7.2f\n", y);
      printf("%-7.2f\n", y);
      printf("%07.2f\n", y);
      printf("%*.*f", 7, 2, y);
      printf("\n");
      printf("%10.2e\n", y);
      printf("%12.4e\n", -y);
      printf("%-10.2e\n", y);
      printf("%e\n", y);
    }
Output
      98.7654
      98.765404
      98.77
      98.77
      0098.77
      98.77
      9.88e+001
      -9.8765e+001
      9.88e+001
      9.876540e+001
```

Fig. 4.10 *Formatted output of* **real** *numbers*

Printing of a Single Character

A single character can be displayed in a desired position using the format:

%wc

The character will be displayed *right-justified* in the field of *w* columns. We can make th display *left-justified* by placing a minus sign before the integer w. The default value for w is 1

Printing of Strings

The format specification for outputting strings is similar to that of real numbers. It is of th form

%w.ps

where *w* specifies the field width for display and *p* instructs that only the first p character of the string are to be displayed. The display is *right-justified*.

The following examples show the effect of variety of specifications in printing a string "NEW DELHI 110001", containing 16 characters (including blanks).

Specification	Output
	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
%s	N E W D E L H I 1 1 0 0 0 1
%20s	N E W D E L H I 1 1 0 0 0 1
%20.10s	N E W D E L H I
%.5s	N E W D
%-20.10s	N E W D E L H I
%5s	N E W D E L H I 1 1 0 0 0 1

Example 4.11 Printing of characters and strings is illustrated in Fig. 4.11.

```
Program
        main()
        {
          char x = 'A';
          char name[20] = "ANIL KUMAR GUPTA";
          printf("OUTPUT OF CHARACTERS\n\n");
          printf("%c\n%3c\n%5c\n", x,x,x);
          printf("%3c\n%c\n", x,x);
```

```
    printf("\n");
    printf("OUTPUT OF STRINGS\n\n");
    printf("%s\n", name);
    printf("%20s\n", name);
    printf("%20.10s\n", name);
    printf("%.5s\n", name);
    printf("%-20.10s\n", name);
    printf("%5s\n", name);
}
Output
OUTPUT OF CHARACTERS
A
 A
  A
 A
A
OUTPUT OF STRINGS
ANIL KUMAR GUPTA
  ANIL KUMAR GUPTA
      ANIL KUMAR
ANIL
ANIL KUMAR
ANIL KUMAR GUPTA
```

Fig. 4.11 *Printing of characters and strings*

Mixed Data Output

It is permitted to mix data types in one **printf** statement. For example, the statement of the type

```
printf("%d %f %s %c", a, b, c, d);
```

is valid. As pointed out earlier, **printf** uses its control string to decide how many variables to be printed and what their types are. Therefore, the format specifications should match the variables in number, order, and type. If there are not enough variables or if they are of the wrong type, the output results will be incorrect.

Table 4.3 *Commonly used printf Format Codes*

Code	*Meaning*
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form
%f	print a floating point value without exponent
%g	print a floating point value either e-type or f-type depending on

Code	Meaning
%i	print a signed decimal integer
%o	print an octal integer, without leading zero
%s	print a string
%u	print an unsigned decimal integer
%x	print a hexadecimal integer, without leading Ox

The following letters may be used as prefix for certain conversion characters.

h for short integers
l for long integers or double
L for long double.

Table 4.4 *Commonly used* **Output** *Format Flags*

Flag	Meaning
–	Output is left-justified within the field. Remaining field will be blank.
+	+ or – will precede the signed numeric item.
0	Causes leading zeros to appear.
# (with o or x)	Causes octal and hex items to be preceded by O and Ox, respectively.
# (with e, f or g)	Causes a decimal point to be present in all floating point numbers, even if it is whole number. Also prevents the truncation of trailing zeros in g-type conversion.

NOTE: C99 adds some more format codes. See the Appendix " C99 Features".

Enhancing the Readability of Output

Computer outputs are used as information for analysing certain relationships between variables and for making decisions. Therefore the correctness and clarity of outputs are of utmost importance. While the correctness depends on the solution procedure, the clarity depends on the way the output is presented. Following are some of the steps we can take to improve the clarity and hence the readability and understandability of outputs.

1. Provide enough blank space between two numbers.
2. Introduce appropriate headings and variable names in the output.
3. Print special messages whenever a peculiar condition occurs in the output.
4. Introduce blank lines between the important sections of the output.

The system usually provides two blank spaces between the numbers. However, this can be increased by selecting a suitable field width for the numbers or by introducing a 'tab' character between the specifications. For example, the statement

```
printf("a = %d\t b = %d", a, b);
```

will provide four blank spaces between the two fields. We can also print them on two separate lines by using the statement

```
printf("a = %d\n b = %d", a, b);
```

Messages and headings can be printed by using the character strings directly in the **printf** statement. Examples:

```
printf("\n OUTPUT RESULTS \n");
printf("Code\t Name\t Age\n");
printf("Error in input data\n");
printf("Enter your name\n");
```

Just Remember

- While using **getchar** function, care should be exercised to clear any unwanted characters in the input stream.
- Do not forget to include **<stdio.h>** headerfiles when using functions from standard input/output library.
- Do not forget to include **<ctype.h>** header file when using functions from character handling library.
- Provide proper field specifications for every variable to be read or printed.
- Enclose format control strings in double quotes.
- Do not forget to use address operator & for basic type variables in the input list of **scanf**.
- Use double quotes for character string constants.
- Use single quotes for single character constants.
- Provide sufficient field with to handle a value to be printed.
- Be aware of the situations where output may be imprecise due to formatting.
- Do not specify any precision in input field specifications.
- Do not provide any white-space at the end of format string of a **scanf** statement.
- Do not forget to close the format string in the **scanf** or **printf** statement with double quotes.
- Using an incorrect conversion code for data type being read or written will result in runtime error.
- Do not forget the comma after the format string in **scanf** and **printf** statements.
- Not separating read and write arguments is an error.
- Do not use commas in the format string of a **scanf** statement.
- Using an address operator & with a variable in the **printf** statement will result in runtime error.

Case Studies

1. Inventory Report

Problem: The ABC Electric Company manufactures four consumer products. Their inventory position on a particular day is given below:

Code	Quantity	Rate (Rs)
F105	275	575.00
H220	107	99.95
I019	321	215.50
M315	89	725.00

It is required to prepare the inventory report table in the following format:

INVENTORY REPORT

Code	Quantity	Rate	Value
———	———	———	———
———	———	———	———
———	———	———	———
———	———	———	———
		Total Value:	———

The value of each item is given by the product of quantity and rate.

Program: The program given in Fig. 4.12 reads the data from the terminal and generates the required output. The program uses subscripted variables which are discussed in Chapter 7.

```
Program
    #define ITEMS 4
    main()
    { /* BEGIN */
      int i, quantity[5];
      float rate[5], value, total_value;
      char code[5][5];
      /* READING VALUES */
      i = 1;
      while ( i <= ITEMS)
      {
        printf("Enter code, quantity, and rate:");
        scanf("%s %d %f", code[i], &quantity[i],&rate[i]);
        i++;
      }
    /*.......Printing of Table and Column Headings.......*/
      printf("\n\n");
      printf("       INVENTORY REPORT       \n");
      printf("-----------------------\n");
      printf("  Code Quantity Rate Value  \n");
      printf("-----------------------\n");
    /*.......Preparation of Inventory Position.....*/
      total_value = 0;
      i = 1;
      while ( i <= ITEMS)
      {
```

```
        value = quantity[i] * rate[i];
        printf("%5s %10d %10.2f %e\n",code[i],quantity[i],
          rate[i],value);
        total_value += value;
        i++;
      }
    /*.......Printing of End of Table.................*/
      printf("-----------------\n");
      printf("          Total Value = %e\n",total_value);
      printf("-----------------\n");
    } /* END */
```

Output

```
Enter code, quantity, and rate:F105 275 575.00
Enter code, quantity, and rate:H220 107 99.95
Enter code, quantity, and rate:I019 321 215.50
Enter code, quantity, and rate:M315 89 725.00
```

INVENTORY REPORT

Code	Quantity	Rate	Value
F105	275	575.00	1.581250e+005
H220	107	99.95	1.069465e+004
I019	321	215.50	6.917550e+004
M315	89	725.00	6.452500e+004
		Total Value =	3.025202e+005

Fig. 4.12 *Program for inventory report*

2. Reliability Graph

Problem: The reliability of an electronic component is given by

$$\text{reliability (r)} = e^{-\lambda t}$$

where λ is the component failure rate per hour and t is the time of operation in hours. A graph is required to determine the reliability at various operating times, from 0 to 3000 hours. The failure rate λ (lambda) is 0.001.

Problem

```
#include <math.h>
#define LAMBDA 0.001
main()
{
  double t;
  float r;
  int i, R;
  for (i=1; i<=27; ++i)
  {
```

```
        printf("--");
    }
    printf("\n");
    for (t=0; t<=3000; t+=150)
    {
        r = exp(-LAMBDA*t);
        R = (int)(50*r+0.5);
        printf(" |");
        for (i=1; i<=R; ++i)
        {
            printf("*");
        }
        printf("#\n");
    }
    for (i=1; i<3; ++i)
    {
        printf(" |\n");
    }
}
```

Output

```
|**************************************************#
|*********************************************#
|*****************************************#
|*********************************#
|****************************#
|************************#
|********************#
|****************#
|**************#
|************#
|**********#
|********#
|******#
|*****#
|*****#
|****#
|***#
|***#
|**#
```

Fig. 4.13 *Program to draw reliability graph*

Program: The program given in Fig. 4.13 produces a shaded graph. The values of t are self-generated by the **for** statement

```
for (t=0; t <= 3000; t = t+150)
```

in steps of 150. The integer 50 in the statement

```
R = (int)(50*r+0.5)
```

is a scale factor which converts r to a large value where an integer is used for plotting the curve. Remember r is always less than 1.

Review Questions

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

(a) The purpose of the header file <studio.h> is to store the programs created by the users.
(b) The C standard function that receives a single character from the keyboard is **getchar**.
(c) The **getchar** cannot be used to read a line of text from the keyboard.
(d) The input list in a **scanf** statement can contain one or more variables.
(e) When an input stream contains more data items than the number of specifications in a **scanf** statement, the unused items will be used by the next **scanf** call in the program.
(f) Format specifiers for output convert internal representations for data to readable characters.
(g) Variables form a legal element of the format control string of a **printf** statement.
(h) The **scanf** function cannot be used to read a single character from the keyboard.
(i) The format specification %+ –8d prints an integer left-justified in a field width of 8 with a plus sign, if the number is positive.
(j) If the field width of a format specifier is larger than the actual width of the value, the value is printed right-justified in the field.
(k) The print list in a **printf** statement can contain function calls.
(l) The format specification %5s will print only the first 5 characters of a given string to be printed.

4.2 Fill in the blanks in the following statements.

(a) The _____specification is used to read or write a short integer.
(b) The conversion specifier _____is used to print integers in hexadecimal form.
(c) For using character functions, we must include the header file _____ in the program.
(d) For reading a double type value, we must use the specification _____.
(e) The specification _____is used to read a data from input list and discard it without assigning it to many variable.
(f) The specification _____ may be used in **scanf** to terminate reading at the encounter of a particular character.
(g) The specification %[ ] is used for reading strings that contain _____.
(h) By default, the real numbers are printed with a precision of _____decimal places.

(i) To print the data left-justified, we must use _____in the field specification.
(j) The specifier _____ prints floating-point values in the scientific notation.

4.3 Distinguish between the following pairs:
(a) ***getchar*** and ***scanf*** functions.
(b) %s and %c specifications for reading.
(c) %s and %[ ] specifications for reading.
(d) %g and %f specification for printing.
(e) %f and %e specifications for printing.

4.4 Write **scanf** statements to read the following data lists:
(a) 78 B 45
(b) 123 1.23 45A
(c) 15-10-2002
(d) 10 TRUE 20

4.5 State the outputs produced by the following **printf** statements.
(a) printf ("%d%c%f", 10, 'x', 1.23);
(b) printf ("%2d %c %4.2f", 1234,, 'x', 1.23);
(c) printf ("%d\t%4.2f", 1234, 456);
(d) printf ("\"%08.2f\"", 123.4);
(e) printf ("%d%d %d", 10, 20);

For questions 4.6 to 4.10 assume that the following declarations have been made in the program:

```
int year, count;
float amount, price;
char code, city[10];
double root;
```

4.6 State errors, if any, in the following input statements.
(a) scanf("%c%f%d", city, &price, &year);
(b) scanf("%s%d", city, amount);
(c) scanf("%f, %d, &amount, &year);
(d) scanf(\n"%f", root);
(e) scanf("%c %d %ld", *code, &count, Root);

4.7 What will be the values stored in the variables **year** and **code** when the data

1988, x

is keyed in as a response to the following statements:
(a) scanf("%d %c", &year, &code);
(b) scanf("%c %d", &year, &code);
(c) scanf("%d %c", &code, &year);
(d) scanf("%s %c", &year, &code);

4.8 The variables **count, price**, and **city** have the following values:

count <—— 1275
price <—— –235.74
city <—— Cambridge

Show the exact output that the following output statements will produce:
(a) printf("%d %f", count, price);
(b) printf("%2d\n%f", count, price);
(c) printf("%d %f", price, count);
(d) `printf("%10dxxxx%5.2f",count, price);`

(e) printf("%s", city);
(f) printf(%-10d %-15s", count, city);

4.9 State what (if anything) is wrong with each of the following output statements:
(a) printf(%d 7.2%f, year, amount);
(b) printf("%-s, %c"\n, city, code);
(c) printf("%f, %d, %s, price, count, city);
(d) printf("%c%d%f\n", amount, code, year);

4.10 In response to the input statement
scanf("%4d%*%d", &year, &code, &count);
the following data is keyed in:
19883745
What values does the computer assign to the variables **year, code**, and **count**?

4.11 How can we use the **getchar( )** function to read multicharacter strings?

4.12 How can we use the **putchar ( )** function to output multicharacter strings?

4.13 What is the purpose of **scanf( )** function?

4.14 Describe the purpose of commonly used conversion characters in a **scanf( )** function.

4.15 What happens when an input data item contains
(a) more characters than the specified field width and
(b) fewer characters than the specified field width?

4.16 What is the purpose of **print( )** function?

4.17 Describe the purpose of commonly used conversion characters in a **printf( )** function.

4.18 How does a control string in a **printf( )** function differ from the control string in a **scanf( )** function?

4.19 What happens if an output data item contains
(a) more characters than the specified field width and
(b) fewer characters than the specified field width?

4.20 How are the unrecognized characters within the control string are interpreted in
(a) **scanf** function; and
(b) **printf** function?

Programming Exercises

4.1 Given the string "WORDPROCESSING", write a program to read the string from the terminal and display the same in the following formats:
(a) WORD PROCESSING
(b) WORD
PROCESSING
(c) W.P.

4.2 Write a program to read the values of x and y and print the results of the following expressions in one line:

(a) $\frac{x+y}{x-y}$ (b) $\frac{x+y}{2}$ (c) $(x+y)(x-y)$

4.3 Write a program to read the following numbers, round them off to the nearest integers and print out the results in integer form:

35.7 50.21 −23.73 −46.45

4.4 Write a program that reads 4 floating point values in the range, 0.0 to 20.0, and prints a horizontal bar chart to represent these values using the character * as the fill character. For the purpose of the chart, the values may be rounded off to the nearest integer. For example, the value 4.36 should be represented as follows.

```
*   *   *   *
*   *   *   *   4.36
*   *   *   *
```

Note that the actual values are shown at the end of each bar.

4.5 Write an interactive program to demonstrate the process of multiplication. The program should ask the user to enter two two-digit integers and print the product of integers as shown below.

```
                    45
                 ×  37
                ------
7 × 45 is          315
3 × 45 is         135
                ------
Add them          1665
                ------
```

4.6 Write a program to read three integers from the keyboard using one **scanf** statement and output them on one line using:

(a) three **printf** statements,
(b) only one **printf** with conversion specifiers, and
(c) only one **printf** without conversion specifiers.

4.7 Write a program that prints the value 10.45678 in exponential format with the following specifications:

(a) correct to two decimal places;
(b) correct to four decimal places; and
(c) correct to eight decimal places.

4.8 Write a program to print the value 345.6789 in fixed-point format with the following specifications:

(a) correct to two decimal places;
(b) correct to five decimal places; and
(c) correct to zero decimal places.

4.9 Write a program to read the name ANIL KUMAR GUPTA in three parts using the **scanf** statement and to display the same in the following format using the **printf** statement.

(a) ANIL K. GUPTA
(b) A.K. GUPTA
(c) GUPTA A.K.

4.10 Write a program to read and display the following table of data.

Name	Code	Price
Fan	67831	1234.50
Motor	450	5786.70

The name and code must be left-justified and price must be right-justified.