11 Console Input/Output

- Types of I/O
- Console I/O Functions
    - Formatted Console I/O Functions
    - *sprintf*( ) and *sscanf*( ) Functions
    - Unformatted Console I/O Functions
- Summary
- Exercise

As mentioned in the first chapter, Dennis Ritchie wanted C to remain compact. In keeping with this intention he deliberately omitted everything related with Input/Output (I/O) from his definition of the language. Thus, C simply has no provision for receiving data from any of the input devices (like say keyboard, disk, etc.), or for sending data to the output devices (like say VDU, disk, etc.). Then how do we manage I/O, and how is it that we were we able to use **printf( )** and **scanf( )** if C has nothing to offer for I/O? This is what we intend to explore in this chapter.

Types of I/O

Though C has no provision for I/O, it of course has to be dealt with at some point or the other. There is not much use writing a program that spends all its time telling itself a secret. Each Operating System has its own facility for inputting and outputting data from and to the files and devices. It's a simple matter for a system programmer to write a few small programs that would link the C compiler for particular Operating system's I/O facilities.

The developers of C Compilers do just that. They write several standard I/O functions and put them in libraries. These libraries are available with all C compilers. Whichever C compiler you are using it's almost certain that you have access to a library of I/O functions.

Do understand that the I/O facilities with different operating systems would be different. Thus, the way one OS displays output on screen may be different than the way another OS does it. For example, the standard library function **printf( )** for DOS-based C compiler has been written keeping in mind the way DOS outputs characters to screen. Similarly, the **printf( )** function for a Unix-based compiler has been written keeping in mind the way Unix outputs characters to screen. We as programmers do not have to bother about which **printf( )** has been written in what manner. We should just use **printf( )** and it would take care of the rest of the

details that are OS dependent. Same is true about all other standard library functions available for I/O.

There are numerous library functions available for I/O. These can be classified into three broad categories:

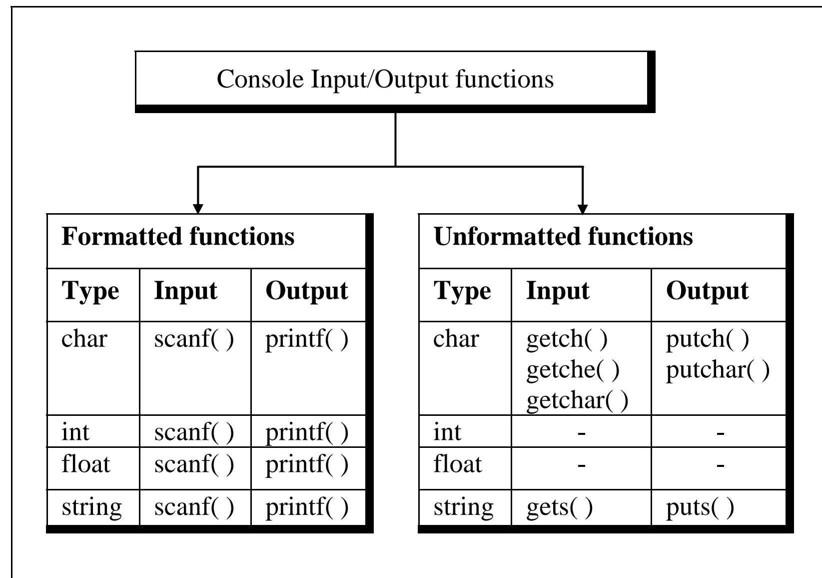
(a) Console I/O functions - Functions to receive input from keyboard and write output to VDU.

(b) File I/O functions - Functions to perform I/O operations on a floppy disk or hard disk.

In this chapter we would be discussing only Console I/O functions. File I/O functions would be discussed in Chapter 12.

Console I/O Functions

The screen and keyboard together are called a console. Console I/O functions can be further classified into two categories—formatted and unformatted console I/O functions. The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points, etc. can be controlled using formatted functions. The functions available under each of these two categories are shown in Figure 11.1. Now let us discuss these console I/O functions in detail.


Console Input/Output functions

Formatted functions		
Type	**Input**	**Output**
char	scanf( )	printf( )
int	scanf( )	printf( )
float	scanf( )	printf( )
string	scanf( )	printf( )

Unformatted functions		
Type	**Input**	**Output**
char	getch( ) getche( ) getchar( )	putch( ) putchar( )
int	-	-
float	-	-
string	gets( )	puts( )

Figure 11.1

Formatted Console I/O Functions

As can be seen from Figure 11.1 the functions **printf( ),** and **scanf( )** fall under the category of formatted console I/O functions. These functions allow us to supply the input in a fixed format and let us obtain the output in the specified form. Let us discuss these functions one by one.

We have talked a lot about **printf( )**, used it regularly, but without having introduced it formally. Well, better late than never. Its general form looks like this...

```
printf ( "format string", list of variables ) ;
```

The format string can contain:

(a) Characters that are simply printed as they are
(b) Conversion specifications that begin with a % sign

(c) Escape sequences that begin with a \ sign

For example, look at the following program:

```
main( )
{
    int  avg = 346 ;
    float  per = 69.2 ;
    printf ( "Average = %d\nPercentage = %f", avg, per ) ;
}
```

The output of the program would be...

```
Average = 346
Percentage = 69.200000
```

How does **printf( )** function interpret the contents of the format string. For this it examines the format string from left to right. So long as it doesn't come across either a **%** or a \ it continues to dump the characters that it encounters, on to the screen. In this example **Average =** is dumped on the screen. The moment it comes across a conversion specification in the format string it picks up the first variable in the list of variables and prints its value in the specified format. In this example, the moment **%d** is met the variable **avg** is picked up and its value is printed. Similarly, when an escape sequence is met it takes the appropriate action. In this example, the moment **\n** is met it places the cursor at the beginning of the next line. This process continues till the end of format string is not reached.

Format Specifications

The **%d** and **%f** used in the **printf( )** are called format specifiers. They tell **printf( )** to print the value of **avg** as a decimal integer and the value of per as a float. Following is the list of format specifiers that can be used with the **printf( )** function.

Data type		Format specifier
Integer	short signed short unsigned long singed long unsigned unsigned hexadecimal unsigned octal	%d or %I %u %ld %lu %x %o
Real	float double	%f %lf
Character	signed character unsigned character	%c %c
String		%s

Figure 11.2

We can provide following optional specifiers in the format specifications.

Specifier	Description
dd	Digits specifying field width
.	Decimal point separating field width from precision (precision stands for the number of places after the decimal point)
dd	Digits specifying precision
-	Minus sign for left justifying the output in the specified field width

Figure 11.3

Now a short explanation about these optional format specifiers. The field-width specifier tells **printf( )** how many columns on screen should be used while printing a value. For example, **%10d** says, “print the variable as a decimal integer in a field of 10 columns”. If the value to be printed happens not to fill up the entire field, the value is right justified and is padded with blanks on the left. If we include the minus sign in format specifier (as in **%-10d**), this means left justification is desired and the value will be padded with blanks on the right. Here is an example that should make this point clear.

```
main( )
{
    int  weight = 63 ;

    printf ( "\nweight is %d kg", weight ) ;
    printf ( "\nweight is %2d kg", weight ) ;
    printf ( "\nweight is %4d kg", weight ) ;
    printf ( "\nweight is %6d kg", weight ) ;
    printf ( "\nweight is %-6d kg", weight ) ;
}
```

The output of the program would look like this ...

```
Columns     0123456789012345678901234567890
            weight   is 63 kg
            weight   is 63 kg
            weight   is 63  kg
            weight   is 63 kg
            weight   is 63 kg
```

Specifying the field width can be useful in creating tables of numeric values, as the following program demonstrates.

```
main( )
{
    printf ( "\n%f %f %f", 5.0, 13.5, 133.9 ) ;
```

```
    printf ( "\n%f %f %f", 305.0, 1200.9, 3005.3 ) ;
}
```

And here is the output...

```
5.000000 13.500000 133.900000
305.000000 1200.900000 3005.300000
```

Even though the numbers have been printed, the numbers have not been lined up properly and hence are hard to read. A better way would be something like this...

```
main( )
{
    printf ( "\n%10.1f %10.1f %10.1f", 5.0, 13.5, 133.9 ) ;
    printf ( "\n%10.1f %10.1f %10.1f", 305.0, 1200.9, 3005.3 );
}
```

This results into a much better output...

```
01234567890123456789012345678901
       5.0       13.5      133.9
     305.0     1200.9     3005.3
```

The format specifiers could be used even while displaying a string of characters. The following program would clarify this point:

```
/* Formatting strings with printf( ) */
main( )
{
    char  firstname1[ ] = "Sandy" ;
    char  surname1[ ] = "Malya" ;
    char  firstname2[ ] = "AjayKumar" ;
    char  surname2[ ] = "Gurubaxani" ;

    printf ( "\n%20s%20s", firstname1, surname1 ) ;
    printf ( "\n%20s%20s", firstname2, surname2 ) ;
```

```
}
```

And here's the output...

```
01234567890123456789012345678901234567890123456789 0
                    Sandy                    Malya
              AjayKumar               Gurubaxani
```

The format specifier **%20s** reserves 20 columns for printing a string and then prints the string in these 20 columns with right justification. This helps lining up names of different lengths properly. Obviously, the format **%-20s** would have left justified the string.

Escape Sequences

We saw earlier how the newline character, **\n**, when inserted in a **printf( )**'s format string, takes the cursor to the beginning of the next line. The newline character is an 'escape sequence', so called because the backslash symbol (\) is considered as an 'escape' character—it causes an escape from the normal interpretation of a string, so that the next character is recognized as one having a special meaning.

The following example shows usage of **\n** and a new escape sequence \t, called 'tab'. A **\t** moves the cursor to the next tab stop. A 80-column screen usually has 10 tab stops. In other words, the screen is divided into 10 zones of 8 columns each. Printing a tab takes the cursor to the beginning of next printing zone. For example, if cursor is positioned in column 5, then printing a tab takes it to column 8.

```
main( )
{
    printf ( "You\tmust\tbe\tcrazy\nto\thate\tthis\tbook" ) ;
}
```

And here's the output...

```
              1             2            3            4
01234567890123456789012345678901234567890
You        must       be        crazy
to         hate       this      book
```

The **\n** character causes a new line to begin following 'crazy'. The tab and newline are probably the most commonly used escape sequences, but there are others as well. Figure 11.4 shows a complete list of these escape sequences.

Esc. Seq.	Purpose	Esc. Seq.	Purpose
\n	New line	\t	Tab
\b	Backspace	\r	Carriage return
\f	Form feed	\a	Alert
\'	Single quote	\"	Double quote
\\	Backslash		

Figure 11.4

The first few of these escape sequences are more or less self-explanatory. **\b** moves the cursor one position to the left of its current position. **\r** takes the cursor to the beginning of the line in which it is currently placed. **\a** alerts the user by sounding the speaker inside the computer. Form feed advances the computer stationery attached to the printer to the top of the next page. Characters that are ordinarily used as delimiters... the single quote, double quote, and the backslash can be printed by preceding them with the backslash. Thus, the statement,

```
printf ( "He said, \"Let's do it!\"" ) ;
```

will print...

```
He said, "Let's do it!"
```

So far we have been describing **printf( )'**s specification as if we are forced to use only **%d** for an integer, only **%c** for a char, only **%s** for a string and so on. This is not true at all. In fact, **printf( )** uses the specification that we mention and attempts to perform the specified conversion, and does its best to produce a proper result. Sometimes the result is nonsensical, as in case when we ask it to print a string using **%d**. Sometimes the result is useful, as in the case we ask **printf( )** to print ASCII value of a character using **%d**. Sometimes the result is disastrous and the entire program blows up.

The following program shows a few of these conversions, some sensible, some weird.

```
main( )
{
    char  ch = 'z' ;
    int  i = 125 ;
    float  a = 12.55 ;
    char  s[ ] = "hello there !" ;

    printf ( "\n%c %d %f", ch, ch, ch ) ;
    printf ( "\n%s %d %f", s, s, s ) ;
    printf ( "\n%c %d %f",i ,i, i ) ;
    printf ( "\n%f %d\n", a, a ) ;
}
```

And here's the output ...

```
z 122 -9362831782501783000000000000000000000000000.000000
hello there ! 3280 -
9362831782501783000000000000000000000000000.000000
} 125 -9362831782501783000000000000000000000000000.000000
```

```
12.550000 0
```

I would leave it to you to analyze the results by yourselves. Some of the conversions you would find are quite sensible.

Let us now turn our attention to **scanf( )**. **scanf( )** allows us to enter data from keyboard that will be formatted in a certain way.

The general form of **scanf( )** statement is as follows:

```
scanf ( "format string", list of addresses of variables ) ;
```

For example:

```
scanf ( "%d %f %c", &c, &a, &ch ) ;
```

Note that we are sending addresses of variables (addresses are obtained by using '**&**' the 'address of' operator) to **scanf( )** function. This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses. The values that are supplied through the keyboard must be separated by either blank(s), tab(s), or newline(s). Do not include these escape sequences in the format string.

All the format specifications that we learnt in **printf( )** function are applicable to **scanf( )** function as well.

sprintf( ) and *sscanf( )* Functions

The **sprintf( )** function works similar to the **printf( )** function except for one small difference. Instead of sending the output to the screen as **printf( )** does, this function writes the output to an array of characters. The following program illustrates this.

```
main( )
{
```

```
    int  i = 10 ;
    char  ch = 'A' ;
    float  a = 3.14 ;
    char  str[20] ;

    printf ( "\n%d %c %f", i, ch, a ) ;
    sprintf ( str, "%d %c %f", i, ch, a ) ;
    printf ( "\n%s", str ) ;
}
```

In this program the **printf( )** prints out the values of **i**, **ch** and **a** on the screen, whereas **sprintf( )** stores these values in the character array **str**. Since the string **str** is present in memory what is written into **str** using **sprintf( )** doesn't get displayed on the screen. Once **str** has been built, its contents can be displayed on the screen. In our program this was achieved by the second **printf( )** statement.

The counterpart of **sprintf( )** is the **sscanf( )** function. It allows us to read characters from a string and to convert and store them in C variables according to specified formats. The **sscanf( )** function comes in handy for in-memory conversion of characters to values. You may find it convenient to read in strings from a file and then extract values from a string by using **sscanf( )**. The usage of **sscanf( )** is same as **scanf( )**, except that the first argument is the string from which reading is to take place.

Unformatted Console I/O Functions

There are several standard library functions available under this category—those that can deal with a single character and those that can deal with a string of characters. For openers let us look at those which handle one character at a time.

So far for input we have consistently used the **scanf( )** function. However, for some situations the **scanf( )** function has one glaring weakness... you need to hit the Enter key before the function can

digest what you have typed. However, we often want a function that will read a single character the instant it is typed without waiting for the Enter key to be hit. **getch( )** and **getche( )** are two functions which serve this purpose. These functions return the character that has been most recently typed. The 'e' in **getche( )** function means it echoes (displays) the character that you typed to the screen. As against this **getch( )** just returns the character that you typed without echoing it on the screen. **getchar( )** works similarly and echo's the character that you typed on the screen, but unfortunately requires Enter key to be typed following the character that you typed. The difference between **getchar( )** and **fgetchar( )** is that the former is a macro whereas the latter is a function. Here is a sample program that illustrates the use of these functions.

```
main( )
{
    char  ch ;

    printf ( "\nPress any key to continue" ) ;
    getch( ) ;  /* will not echo the character */

    printf ( "\nType any character" ) ;
    ch = getche( ) ;  /* will echo the character typed */

    printf ( "\nType any character" ) ;
    getchar( ) ;  /* will echo character, must be followed by enter key */
    printf ( "\nContinue Y/N" ) ;
    fgetchar( ) ;  /* will echo character, must be followed by enter key */
}
```

And here is a sample run of this program...

```
Press any key to continue
Type any character B
Type any character W
Continue Y/N Y
```

putch( ) and **putchar( )** form the other side of the coin. They print a character on the screen. As far as the working of **putch( ) putchar( )** and **fputchar( )** is concerned it's exactly same. The following program illustrates this.

```
main( )
{
    char  ch = 'A' ;

    putch ( ch ) ;
    putchar ( ch ) ;
    fputchar ( ch ) ;
    putch ( 'Z' ) ;
    putchar ( 'Z' ) ;
    fputchar ( 'Z' ) ;
}
```

And here is the output...

```
AAAZZZ
```

The limitation of **putch( ), putchar( )** and **fputchar( )** is that they can output only one character at a time.

gets( )* and *puts( )

gets( ) receives a string from the keyboard. Why is it needed? Because **scanf( )** function has some limitations while receiving string of characters, as the following example illustrates...

```
main( )
{
    char  name[50] ;

    printf ( "\nEnter name " ) ;
    scanf ( "%s", name ) ;
    printf ( "%s", name ) ;
```

```
}
```

And here is the output...

```
Enter name Jonty Rhodes
Jonty
```

Surprised? Where did “Rhodes” go? It never got stored in the array **name[ ],** because the moment the blank was typed after “Jonty” **scanf( )** assumed that the name being entered has ended. The result is that there is no way (at least not without a lot of trouble on the programmer’s part) to enter a multi-word string into a single variable (**name** in this case) using **scanf( ).** The solution to this problem is to use **gets( )** function. As said earlier, it gets a string from the keyboard. It is terminated when an Enter key is hit. Thus, spaces and tabs are perfectly acceptable as part of the input string. More exactly, **gets( )** gets a newline (**\n**) terminated string of characters from the keyboard and replaces the **\n** with a **\0**.

The **puts( )** function works exactly opposite to **gets( )** function. It outputs a string to the screen.

Here is a program which illustrates the usage of these functions:

```
main( )
{
    char  footballer[40] ;

    puts ( "Enter name" ) ;
    gets ( footballer ) ;  /* sends base address of array */
    puts ( "Happy footballing!" ) ;
    puts ( footballer ) ;
}
```

Following is the sample output:

```
Enter name
```

```
Jonty Rhodes
Happy footballing!
Jonty Rhodes
```

Why did we use two **puts( )** functions to print "Happy footballing!" and "Jonty Rhodes"? Because, unlike **printf( ), puts( )** can output only one string at a time. If we attempt to print two strings using **puts( ),** only the first one gets printed. Similarly, unlike **scanf( ), gets( )** can be used to read only one string at a time.

Summary

(a) There is no keyword available in C for doing input/output.
(b) All I/O in C is done using standard library functions.
(c) There are several functions available for performing console input/output.
(d) The formatted console I/O functions can force the user to receive the input in a fixed format and display the output in a fixed format.
(e) There are several format specifiers and escape sequences available to format input and output.
(f) Unformatted console I/O functions work faster since they do not have the overheads of formatting the input or output.

Exercise

[A] What would be the output of the following programs:

(a)
```
main( )
{
    char ch ;
    ch = getchar( ) ;
    if ( islower ( ch ) )
        putchar ( toupper ( ch ) ) ;
    else
        putchar ( tolower ( ch ) ) ;
```

```
        }
(b)     main( )
        {
            int i = 2 ;
            float f = 2.5367 ;
            char str[ ] = "Life is like that" ;

            printf ( "\n%4d\t%3.3f\t%4s", i, f, str ) ;
        }
(c)     main( )
        {
            printf ( "More often than \b\b not \rthe person who \
                        wins is the one who thinks he can!" ) ;
        }
(d)     char  p[ ] = "The sixth sick sheikh's sixth ship is sick" ;
        main( )
        {
            int  i = 0 ;
            while ( p[i] != '\0' )
            {
                putch ( p[i] ) ;
                i++ ;
            }
        }
```

[B] Point out the errors, if any, in the following programs:

```
(a)     main( )
        {
            int i ;
            char a[ ] = "Hello" ;
            while ( a != '\0' )
            {
                printf ( "%c", *a ) ;
                a++ ;
            }
        }
```

(b)
```
main( )
{
    double dval ;
    scanf ( "%f", &dval ) ;
    printf ( "\nDouble Value = %lf", dval ) ;
}
```

(c)
```
main( )
{
    int ival ;
    scanf ( "%d\n", &n ) ;
    printf ( "\nInteger Value = %d", ival ) ;
}
```

(d)
```
main( )
{
    char  *mess[5] ;
    for ( i = 0 ; i < 5 ; i++ )
        scanf ( "%s", mess[i] ) ;
}
```

(e)
```
main( )
{
    int dd, mm, yy ;
    printf ( "\nEnter day, month and year\n" ) ;
    scanf ( "%d%*c%d%*c%d", &dd, &mm, &yy ) ;
    printf ( "The date is: %d - %d - %d", dd, mm, yy ) ;
}
```

(f)
```
main( )
{
    char text ;
    sprintf ( text, "%4d\t%2.2f\n%s", 12, 3.452, "Merry Go Round" ) ;
    printf ( "\n%s", text ) ;
}
```

(g)
```
main( )
{
    char buffer[50] ;
```

```
    int no = 97;
    double val = 2.34174 ;
    char name[10] = "Shweta" ;

    sprintf ( buffer, "%d %lf %s", no, val, name ) ;
    printf ( "\n%s", buffer ) ;
    sscanf ( buffer, "%4d %2.2lf %s", &no, &val, name ) ;
    printf ( "\n%s", buffer ) ;
    printf ( "\n%d %lf %s", no, val, name ) ;

}
```

[C] Answer the following:

(a) To receive the string "We have got the guts, you get the glory!!" in an array **char str[100]** which of the following functions would you use?

1. scanf ("%s", str) ;
2. gets (str) ;
3. getche (str) ;
4. fgetchar (str) ;

(b) Which function would you use if a single key were to be received through the keyboard?

1. scanf( )
2. gets( )
3. getche( )
4. getchar( )

(c) If an integer is to be entered through the keyboard, which function would you use?

1. scanf( )
2. gets( )
3. getche( )
4. getchar( )

(d) If a character string is to be received through the keyboard which function would work faster?

1. scanf( )
2. gets( )

(e) What is the difference between **getchar( )**, **fgetchar( )**, **getch( )** and **getche( )**?

(f) The format string of a **printf( )** function can contain:

1. Characters, format specifications and escape sequences
2. Character, integers and floats
3. Strings, integers and escape sequences
4. Inverted commas, percentage sign and backslash character

(g) A field-width specifier in a **printf( )** function:

1. Controls the margins of the program listing
2. Specifies the maximum value of a number
3. Controls the size of type used to print numbers
4. Specifies how many columns will be used to print the number

[D] Answer the following:

(a) Write down two functions **xgets( )** and **xputs( )** which work similar to the standard library functions **gets( )** and **puts( )**.

(b) Write down a function **getint( )**, which would receive a numeric string from the keyboard, convert it to an integer number and return the integer to the calling function. A sample usage of **getint( )** is shown below:

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

```
    a = getint( ) ;
    printf ( "you entered %d", a )
}
```