Unit 3 Data Types and Input/Output Operators

Structure:

3.1 Introduction

In the previous unit, you learned about the operators that a C programming language supports. You also learned how the operators are used in the expressions of C programs. In this unit, you will learn about the data types that are supported in C. You will also study about the Input/output operators which makes C as the most efficient and powerful programming language. Integer is one of the fundamental data types. All C compilers support four fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), and double-precision floating point (**double**). Like integer data type, other data types also offer extended data types such as **long double** and **signed char.**

C supports a rich set of operators. We have already used several of them, such as =, +, -, *, / and %. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

It is possible to combine operands of different data types in arithmetic expressions. Such expressions are called mixed-mode arithmetic expressions.

C language is accompanied by some library functions to handle input/output(I/O) operations. In this unit we will make use of six I/O functions : getchar(), putchar(), scanf(), printf(), gets() and puts(). These functions are used to transfer the information between the computer and the standard input/output devices. Throughout this course we assume that keyboard is the standard input device and the user screen is the standard output device. The first two functions, getchar() and putchar(), allow single character to be transferred into and out of the computer; the functions scanf() and printf() permit the transfer of single character, numerical values and strings; the functions gets() and puts() facilitate the input and output of strings. These functions can be accessed within a program by including the header file stdio.h.

Objectives:

After studying this unit, you should be able to:

- apply the concept of real numbers in C and characters in C
- combine different data types and form more complicated arithmetic expressions
- to transfer a character, numerical value and a string between the computer and I/O devices
- to write programs using I/O functions to handle single character, numerical values and strings

3.2 Floating-point Numbers

Floating point (or real) numbers are stored in 32 bit (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. To extend the precision further, we may use **long double** which uses 80 bits. The table 3.1 shows all the allowed combinations of floating point numbers and qualifiers and their size and range on a 16-bit machine.

Table 3.1: Floating point numbers and qualifiers and their size and range on a 16-bit machine.

Type	Size (bits)	Range
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

Program 3.1: The following program illustrates typical declarations, assignments and values stored in various types of variables.

```
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
```

Program 3.2: Program to calculate the average of N numbers

```
#define  N 10  /* SYMBOLIC CONSTANT */
main()
{
        int       count;                        /* DECLARATION OF
        float     sum, average, number;         VARIABLES */
        sum  =  0;                              / * INITIALIZATION OF
        count = 0;                                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
```

Program 3.3: Program to compute the roots of a quadratic equation

```
#include <math.h>
main()
{
        float a,b,c,discriminant, root1, root2;
        printf(“input the values of a,b and c\n”);
        scanf (“%f %f %f”, &a, &b, &c);
        discriminant = b * b – 4 * a *c;
        if (discriminant<0)
                printf(“roots are imaginary\n”);
        else
                {
                root1 =  (-b + sqrt(discriminant)) /  (2 * a);
                root2 =  (-b - sqrt(discriminant)) /  (2 * a);
                printf (“Root1 = %5.2f \n    Root2 = %5.2f \n”, root1, root2);
                }
}
```

Output

```
input the values of a,b and c
2 4 -16
Root1 = 2.00
Root2 = -4.00
input the values of a,b and c
1 2 3
roots are imaginary
```

3.2.1 Converting Integers to Floating-point and vice-versa

C permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. We know that the computer considers one operator at a time, involving two operands.

If the operands are of different types, the ‘lower’ type is automatically converted to the ‘higher’ type before the operation proceeds. The result is of higher type.

Given below is the sequence of rules that are applied while evaluating expressions.

All **short** type are automatically converted to **int ;** then

1. If one of the operands is **long double,** the other will be converted to **long double** and the result will be **long double**;
2. else, if one of the operands is **double**, the other will be converted to **double** and the result will be **double**;
3. else, if one of the operands is **float**, the other will be converted to **float** and the result will be **float**;
4. else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and the result will be **unsigned long int**;
5. else if one of the operands is **long int** and the other is **unsigned int**, then:
   - if **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted as such and the result will be **long int**;
   - else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int**;
6. else, if one of the operands is **long int** , the other will be converted to **long int** and the result will be **long int**;
7. else, if one of the operands is **unsigned int** , the other will be converted to **unsigned int** and the result will be **unsigned int**;

The final result of an expression is converted to type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment:

1. **float** to **int** causes truncation of the fractional part.
2. **double** to **float** causes rounding of digits.
3. **long int** to **int** causes dropping of the excess higher order bits

3.2.2 Mixed-mode Expressions

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of the real type, then only the real operation is performed and the result is always real number. Thus

25 / 10.0 = 2.5

Whereas 25 / 10 =2

Self Assessment Questions

1. When the accuracy provided by a **float** number is not sufficient, the type **long float** can be used to define the number. (True/False)
2. A **double** data type uses __________ bits.
3. If the operands are of different data types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. (True/False)
4. During the final assignment ________ to int causes dropping of the excess higher order bits.
5. The value of the expression 22.0/10 is ________.

3.3 The type cast Operator

C performs type conversions automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of doctors to engineers in a town.

Ratio = doctor_number / engineer _number

Since doctor _number and engineer_number are declared as integers in the program, the decimal part of the result of the division would be lost and Ratio would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

Ratio = (**float**) doctor_number / engineer _number

The operator (**float**) converts the doctor_number to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of the result. Note that in no way does the operator (**float**) affect the value of the variable doctor_number. And also, the type of doctor_number remains as **int** in the other parts of the program.

The process of such local conversion is known as casting a value. The general form of cast is:

(type-name) expression

where type-name is one of the standard C data types. The expression may be a constant, variable or an expression. The Table 3.2 shows some examples of casts and their actions:

Table 3.2: Use of Casts

Example	Action
X=(**int)** 8.5	8.5 is converted to integer by truncation.
A=(**int) 21.3** / **(int)** 4.5	Evaluated as 21/4 and the result would be 5.
B=(**double**) sum/n	Division is done in floating point mode.
Y= (**int**) (a+b)	The result of a+b is converted to integer.
Z= (**int)** a+b	a is converted to integer and then added to b.
P=cos((**double**)x)	Converts x to double before using it.

Program 3.4: The following program shows the use of casts

```
main()
{
        /* Program to find average of two integers */
        float  avg;
        int n=2,n1,n2;
        printf("enter any 2  numbers\n");
        scanf("%d %d",&n1,&n2);
        avg=(n1+n2)/(float)n;
        printf(" their average is\n",avg);
}
```

Casting can be used to round-off a given value. Consider the following statement:

X= (**int**) (y+0.5);

If y is 37.7, y+0.5 is 38.2 and on casting, the result becomes 38, the value that is assigned to X. Of course, the expression, being cast is not changed. When combining two different types of variables in an expression, never assume the rules of automatic conversion. It is always a good practice to explicitly force the conversion. It is more safer and more portable. For example, when **y** and **p** are double and **m** is **int**, the following two statements are equivalent.

y = p + m;

y = p + (double)m;

However, the second statement is preferable. It will work the same way on all machines and is more readable.

Self Assessment Questions

6. Casting can be used to round-off a given value. (True/False)
7. The value of A in the expression A=(int)11.35/(int)14.5 is ___________.
8. If the value of X is 35.2, then the value of A in the expression:
   A = (int)(X+0.5); is ______________.

3.4 The type char

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 char**s have values between 0 and 255, **signed char**s have values from -128 to 127.

A character constant is formed by enclosing the character within a pair of single quote marks. So ‘b’, ‘.’ and ‘5’ are all valid examples of character constants. Note that a character constant, which is a single character enclosed in single quotes is different from a character string, which is any number of characters enclosed in double quotes.

The format characters %c can be used in a printf statement to display the value of a char variable at the terminal.

Program 3.5: The following program illustrates how to use char data type.

```
#include<stdio.h>
main()
{
 char c=’A’;
int a=65;
printf(“%c\n”, c);
printf(“%d\n”, c);
printf(“%c\n”,a);
}
```

Output

```
A
65
A
```

Note that with the format characters %d, the ASCII number of the character is displayed. With the format character %c, the character corresponding to the given ASCII number is displayed.

Self Assessment Questions

9. What is the format character to display the value of a char variable?
10. The output of the following C statement:
    printf("%c", 70); is ___________.

3.5 Keywords

Keywords are the reserved words of a programming language. All the keywords have fixed meanings and these meanings cannot be changed.

Keywords serve as basic building blocks for program statements. The list of all keywords in ANSI C are listed in the Table 3.3.

Table 3.3: 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

All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

Self Assessment Questions

11. All keywords must be written in ____________.
12. **default** is not a valid keyword in C. (True/False)

3.6 Character Input and Output

The most basic way of reading input is by calling the function **getchar(). getchar()** reads one character from the "standard input," which is usually the user's keyboard. **getchar()** returns (rather obviously) the character it reads, or, if there are no more characters available, the special value EOF ("end of file"). This value will be assigned within the stdio.h file. Typically, EOF will be assigned the value -1, but this may vary from one compiler to another.

The syntax of the **getchar()** function is written as

character variable= getchar()

where *character variable* refers to some previously declared character variable.

Example:

```
char c;
  …
c=getchar();
```

The first statement declares that c is a character-type variable. The second statement causes a single character to be entered from the keyboard and then assign to c.

A companion function is **putchar(),** which writes one character to the “standard output." (The standard output is usually the user's screen).

The syntax of the **putchar()** function is written as

putchar(character variable)

where *character variable* refers to some previously declared character variable.

Example:

```
char c;
…
putchar(c);
```

The first statement declares that c is a character-type variable. The second statement causes the current value of c to be transmitted to the user monitor where it will be displayed.

Using these two functions, we can write a very basic program to copy the input, a character at a time, to the output:

Program 3.6: Program to copy the input, a character at a time, to the output

```
#include <stdio.h>
/* copy input to output */
main()
{
        int c;
        c = getchar();
        while(c != EOF)
                {
                putchar(c);
                c = getchar();
                }
        return 0;
}
```

Execute the above program and observe the result.

It reads one character, and if it is not the EOF code, enters a while loop, printing one character and reading another, as long as the character read is not EOF. A **char** variable could hold integers corresponding to character set values, and that an **int** could hold integers of more arbitrary values (from -32768 to + 32767). Since most character sets contain a few hundred characters (nowhere near 32767), an **int** variable can in general comfortably hold all **char** values, and then some. Therefore, there's nothing wrong with declaring c as an **int.** But in fact, it's important to do so, because **getchar()** can return every character value, *plus* that special, non-character value EOF, indicating that there are no more characters. Type **char** is only guaranteed to be able to hold all the character values; it is *not* guaranteed to be able to hold EOF value without possibly mixing it up with some actual character value. Therefore, you should always remember to use an **int** for anything you assign **getchar()**'s return value to.

When you run the character copying program, and it begins copying its input (you're typing) to its output (your screen), you may find yourself wondering how to stop it. It stops when it receives end-of-file (EOF), but how do you send EOF? The answer depends on what kind of computer you're using. On

Unix and Unix-related systems, it's almost always control-D. On MS-DOS machines, it's control-Z followed by the RETURN key.

(Note, too, that the character you type to generate an end-of-file condition from the keyboard is *not* the same as the special EOF value returned by **getchar().** The EOF value returned by **getchar()** is a code indicating that the input system has detected an end-of-file condition, whether it's reading the keyboard or a file or a magnetic tape or a network connection or anything else. In a disk file, at least, there is not likely to be any character in the file corresponding to EOF; as far as your program is concerned, EOF indicates the absence of any more characters to read.)

Another excellent thing to know when doing any kind of programming is how to terminate a runaway program. If a program is running forever waiting for input, you can usually stop it by sending it an end-of-file, as above, but if it's running forever *not* waiting for something, you'll have to take more drastic measures. Under Unix, control-C (or, occasionally, the DELETE key) will terminate the current program, almost no matter what. Under MS-DOS, control-C or control-BREAK will sometimes terminate the current program.

Self Assessment Questions

13. getchar() function is an output function.(True/False)
14. In order to stop reading the input character, you can use a value called __________________.

3.7 Formatted Input and Output

Input data can be entered into the computer from a standard input device by means of the standard C library function **scanf()**. This function can be used to enter any combination of numerical values, single character and strings. The function returns the number of data items that have been entered successfully.

The syntax of scanf function is as follows:

scanf(control string, arg1, arg2, …argn)

where *control string* refers to a string containing certain required formatting information, and *arg1, arg2,…, argn* are arguments that represent the individual input data items. The arguments represent *pointers* that indicate *addresses* of the data items within the computer's memory.

The *control string* consists of control characters, whitespace characters, and non-whitespace characters. The control characters are preceded by a % sign, and are listed in Table 3.4.

Table 3.4: Control characters and its explanations

Control Character	**Explanation**
%c	a single character
%d	a decimal integer
%i	an integer
%e, %f, %g	a floating-point number
%o	an octal number
%s	a string
%x	a hexadecimal number
%p	a pointer
%n	an integer equal to the number of characters read so far
%u	an unsigned integer
%[]	a set of characters
%%	a percent sign

scanf() reads the input, matching the characters from format. When a control character is read, it puts the value in the next variable. Whitespaces (tabs, spaces, etc) are skipped. Non-whitespace characters are matched to the input, then discarded. If a number comes between the % sign and the control character, then only that many characters will be entered into the variable. If **scanf()** encounters a set of characters, denoted by the %[] control character, then any characters found within the brackets are read into the variable. The return value of **scanf()** is the number of variables that were successfully assigned values, or EOF if there is an error.

Program 3.7: Program to use scanf() to read integers, floats, characters and strings from the user.

```
#include<stdio.h>
main()
{
  int i;
 float f;
 char c;
 char str[10];
 scanf("%d %f %c %s", &i, &f, &c, str);
 printf("%d %f %c %s", i, f, c, str);
}
```

Execute this program and observe the result.

Note that for a **scanf()** function, the addresses of the variable are used as the arguments for an **int, float** and a **char** type variable. But this is not true for a string variable because a string name itself refers to the address of a string variable.

A *s-control* character is used to enter strings to string variables. A string that includes whitespace characters can not be entered. There are ways to work with strings that include whitespace characters. One way is to use the **getchar()** function within a loop. Another way is to use **gets()** function which will be discussed later.

It is also possible to use the **scanf()** function to enter such strings. To do so, the s-control character must be replaced by a sequence of characters enclosed in square brackets, designated as […]. Whitespace characters may be included within the brackets, thus accommodating strings that contain such characters.

Example:

```
#include<stdio.h>
main()
{
 char str[80];
 …
 scanf("%[ ABCDEFGHIJKLMNOPQRST]", str);
 …
}
```

This example illustrates the use of the scanf() function to enter a string consisting of uppercase letters and blank spaces. Please note that if you want to allow lowercase letters to be entered, all the lowercase letters (i.e. from a-z) must be included in the list of control string.

Formatted Output

Output data can be written from the computer onto a standard output device using the library function **printf().** This function can be used to output any combination of numerical values, single characters and strings. It is similar to the input function **scanf(),** except that its purpose is to display data rather than enter into the computer.

The syntax of the **printf** function can be written as follows:

printf(control string, arg1, arg2, …, argn)

where *control string* refers to a string that contains formatting information, and arg1, arg2, …, argn are arguments that represent the individual output data items. The arguments can be written as constants, single variable or array names, or more complex expressions.

Examples:

```
printf("Hello, world!\n");
printf("i is %d\n", i);
printf("%d", 10);
printf("%d", i+j);
```

The first statement simply displays the string given as argument to the **printf()** function. In the second statement, **printf()** function replaces the two characters %d with the value of the variable i. In the third statement the argument to be printed is a constant and in the fourth, the argument is an expression.

There are quite a number of format specifiers for printf(). Some of them are listed in Table 3.5.

Table 3.5: Format specifiers for printf().

%d	Print an int argument in decimal
%ld	print a long int argument in decimal
%c	print a character
%s	print a string
%f	print a float or double argument
%e	same as %f, but use exponential notation
%g	use %e or %f, whichever is better
%o	print an int argument in octal (base 8)
%x	print an int argument in hexadecimal (base 16)
%%	print a single %

It is also possible to specify the width and precision of numbers and strings as they are inserted ; For example, a notation like %3d means to print an **int** in a field at least 3 spaces wide; a notation like %5.2f means to print a **float** or **double** in a field at least 5 spaces wide, with two places to the right of the decimal.)

To illustrate with a few more examples: the call

```
printf("%c %d %f %e %s %d%%\n", '3', 4, 3.24, 66000000, "nine", 8);
```

would print

```
3 4 3.240000 6.600000e+07 nine 8%
```

The call

```
printf("%d %o %x\n", 100, 100, 100);
```

would print

```
100 144 64
```

Successive calls to **printf()** just build up the output a piece at a time, so the calls

```
printf("Hello, ");
printf("world!\n");
```

would also print Hello, world! (on one line of output).

Earlier we learned that C represents characters internally as small integers corresponding to the characters' values in the machine's character set (typically ASCII). This means that there isn't really much difference between a character and an integer in C; most of the difference is in whether we choose to interpret an integer as an integer or a character. **printf** is one place where we get to make that choice: %d prints an integer value as a string of digits representing its decimal value, while %c prints the character corresponding to a character set value. So the lines

```
char c = 'A';
int i = 97;
printf("c = %c, i = %d\n", c, i);
```

would print c as the character A and i as the number 97. But if, on the other hand, we called

```
printf("c = %d, i = %c\n", c, i);
```

we'd see the decimal value (printed by %d) of the character 'A', followed by the character (whatever it is) which happens to have the decimal value 97.

You have to be careful when calling **printf().** It has no way of knowing how many arguments you've passed it or what their types are other than by looking for the format specifiers in the format string. If there are more format specifiers (that is, more % signs) than the arguments, or if the arguments have the wrong types for the format specifiers, **printf()** can misbehave badly, often printing nonsense numbers or (even worse) numbers which mislead you into thinking that some other part of your program is broken.

Because of some automatic conversion rules which we haven't covered yet, you have a small amount of latitude in the types of the expressions you pass as arguments to **printf().** The argument for %c may be of type **char** or **int,** and the argument for %d may be of type **char** or **int.** The string argument for %s may be a string constant, an array of characters, or a pointer to some characters. Finally, the arguments corresponding to %e, %f, and %g may be of types **float** or **double.** But other combinations do *not* work reliably: %d will not print a **long int** or a **float** or a **double;** %ld will not print an **int;** %e, %f, and %g will not print an **int.**

Self Assessment Questions

15. The __________ string consists of control characters, whitespace characters, and non-whitespace characters.
16. The control string used to read a hexadecimal character is _______________.
17. scanf() functions needs address of the data item to be read as the argument. (True/False)

18. The output of the following statement is ________________.
    printf("%d %o %x\n", 64, 10, 75);
19. To print an int argument in octal, you can use ____________ format string.
20. The output of the following program segment is ____________.
    int a=97;
    printf(“%c”, a);

3.8 The gets() and puts() functions

gets() and **puts()** functions facilitate the transfer of strings between the computer and the standard input/output devices. Each of these functions accepts a single argument. The argument must be a data item that represents a string (an array of characters). The string may include whitespace characters. In the case of **gets()**, the string will be entered from the keyboard, and will terminate with a newline character (i.e. a string will end when the user presses the RETURN key).

Example: Reading and writing a line of text.

```
#include<stdio.h>
main()
{
  char line[80];
  gets(line);
  puts(line);
}
```

This program uses **gets()** and **puts()** functions rather than **scanf()** and **printf(),** to transfer the line of text into and out of the computer.

Self Assessment Questions

21. **gets()** is a formatted input statement. (True/False)
22. The argument for a **gets()** and **puts()** functions are ____________ variables.
23. Using **gets()** function, you cannot include whitespace characters in the input string. (True/False)

3.9 Interactive Programming

Creating interactive dialog between the computer and the user is a modern style of programming. These dialogs usually involve some form of question-answer interaction, where the computer asks the questions and the user provides the answer, or vice versa.

In C, such dialogs can be created by alternate use of the **scanf()** and **printf()** functions.

Program 3.8: Program to calculate the simple interest

```
#include<stdio.h>
main()
{
   /* Sample interactive program*/
   float principle, rate, time, interest;
  printf(“ Please enter the principle amount: “);
  scanf(“%f”, &principle);
  printf(“ Please enter the rate of interest: “);
  scanf(“%f”, &rate);
  printf(“ Please enter the period of deposit: “);
  scanf(“%f”, &time);
  interest=principle*rate*time/100.0;
  printf(“Principle=%7.2f\n”, principle);
  printf(“Rate of interest=%5.2f\n”,rate);
  printf(“Period of deposit=%5.2f\n”, time);
  printf(“Interest=%7.2f\n”, interest);
}
```

Execute the above program and observe the result.

3.10 Summary

Floating point(or real) numbers are stored in 32 bit (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. Characters are usually stored in 8 bits (one byte) of internal storage. Like integer data type other data types also offer extended data types such as **long double** and **signed char.** C permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. There are instances when we want to force a type conversion in a way that is different from the automatic conversion. That is, by using type cast operator. All keywords have fixed meanings and these meanings cannot be changed.

getchar(), putchar(), scanf(), printf(), gets() and **puts()** are the commonly used input/output functions in C. These functions are used to transfer of information between the computer and the standard input/output devices. **getchar()** and **putchar()** are the two functions to read and write single character. **scanf()** and **printf()** are the two formatted input/output functions. These functions can handle characters, numerical values and strings as well. **gets()** and **puts()** functions are used to handle strings. **scanf(), printf(), gets()** and **puts()** functions are used in interactive programming.

3.11 Terminal Questions

1. Which of the following arithmetic expressions are valid? If valid, give the value of the expression; otherwise give reason.

   a) 7.5 % 3 b) 14 % 3 + 7 %2

   c) 21 % (int) 4.5 d) 15.25 + - 5.0

2. 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;
```

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

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

4. The _______ **chars** have values from -128 to 127.
5. What are the commonly used input/output functions in C? How are they accessed?
6. Distinguish between **getchar()** and **putchar()** functions?
7. When entering a string using **scanf()** function, how can a single string which includes whitespace characters be entered?
8. Distinguish between **gets()** and **scanf()** functions.
9. A C program contains the following statements:

   ```
   #include<stdio.h>
   int i, j, k;
   ```

   Write an appropriate **scanf()** function to enter numerical values for i, j and k assuming
   a) The values for i, j and k will be decimal integers
   b) The value for i will be a decimal integer, j an octal integer and k a hexadecimal integer.
   c) The values for i and j will be hexadecimal integers and k will be an octal integer.

3.12 Answers to Self Assessment Questions

1. False
2. 64
3. True
4. long int
5. 2.2
6. True
7. 0
8. 35
9. %c
10. False
11. Lowercase
12. False
13. False
14. EOF
15. Control
16. %x

17. True
18. 64, 12, 4B
19. %o
20. a
21. False
22. String
23. False

3.13 Answers to Terminal Questions

1. a) invalid, because % can be used only with integers.
   b) valid, answer is 3
   c) valid, answer is 1
   d) valid, answer is 10.25
2. Errors in the following statements
   i) Int x;
      Can be written as
      int x;
   ii) double = p, q
      Can be written as
      double p,q;
   iii) exponent alpha, beta;
      There is no **data type** exponent in C.
   iv) m,n,z:INTEGER
      Can be written as
      int m,n,z;
   v) short char c;
      There is no data type **short char** in C.
   vi) long int m; count;
      Can be written as
      long int m,count;
   vii) long float temp;
      There is no data type **long float** in C
3. 107
4. signed

5. The commonly used input/output functions in C are: **getchar(), putchar(), scanf(), printf(), gets()** and **puts().** These functions can be accessed within a program by including the header file stdio.h.
6. **getchar()** function is used to accept a single character from the keyboard and **putchar()** function is used to display single character on the user's screen.
7. By using control string %[ ].
8. gets() is not the formatted input function but the scanf() function is a formatted input function.
9. a) scanf("%d %d %d", &i, &j, &k);
   b) scanf("%d %o %x", &i, &j, &k);
   c) scanf("%x %x %o", &i, &j, &k);

3.14 Exercises

1. Represent the following numbers using scientific notation:
   a) 0.001 b) -1.5
2. Represent the following scientific numbers into decimal notation: a) 1.0E+2 b) 0.001E-2
3. What is unsigned char? Explain.
4. What is short char? Explain.
5. Distinguish between float and double data types.
6. Write a program to print the factors of a given number.
7. Given the length of a side, write a C program to compute surface area and volume of a cube.
8. Write a program to reverse a number and find sum of the digits.
9. Write a program to print the multiplication table for any given number.
10. Write a program to check whether a given number is palindrome.