3 Functions

Significant Learning Outcomes

After the completion of this chapter, the learner

- identifies the merits of modular programming in problem solving.
- classifies various input output functions for character and string data.
- compares character input functions.
- uses appropriate character and string functions for the I/O operations.
- applies mathematical functions for solving problems.
- uses string functions for the manipulation of string data.
- manipulates character data with predefined character functions.
- implements modular programming by creating functions.
- identifies the role of arguments and compares different methods of function calling.
- recognises the scope of variables and functions in a program.

We discussed some simple programs in the previous chapters. But to solve complex problems, larger programs running into thousands of lines of code are required. Complex problems are divided into smaller ones and programs to solve each of these sub problems are written. In other words, we break the larger programs into smaller sub programs.. In C++, function is a way to divide large programs into smaller sub programs. We are familiar with the function `main()`. We know that it is the essential function in a C++ program. The statements required to solve a problem are given within a pair of braces `{` and `}` after the header `int main()`. We have not broken a problem into sub problems so far and hence the entire task is assisgned to `main()` function. But there are some functions readily available for use, which makes programming simpler. Each of them is assigned with a specific task and they are stored in header files. So, these functions are known as built-in functions or predefined functions. Besides such functions, we can define functions for a specific task. These are called user-defined functions. In this chapter we will discuss some important predefined

functions and learn how to define our own functions. Before going into these, let us familiarise ourselves with a style of programming called modular programming.

3.1 Concept of modular programming

Let us consider the case of a school management software. It is a very large and complex software which may contain many programs for different tasks. The complex task of school management can be divided into smaller tasks or modules and developed in parallel, and later integrated to build the complete software as shown in Figure 3.1.

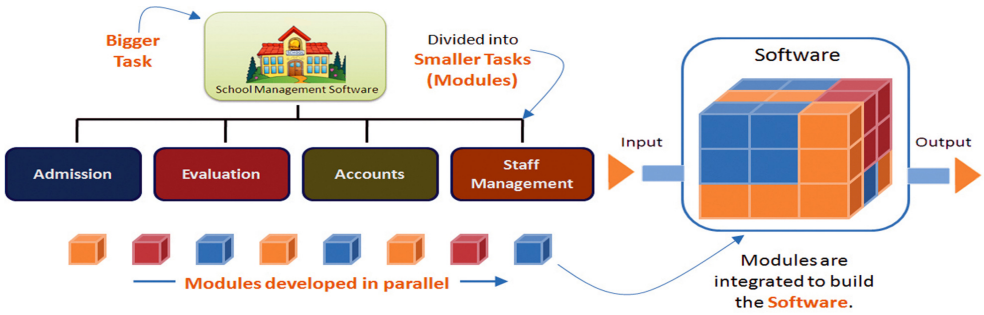

Fig. 31: Modular programming style

In programming, the entire problem will be divided into small sub problems that can be solved by writing separate programs. This kind of approach is known as modular programming. Each sub task will be considered as a module and we write programs for each module. The process of breaking large programs into smaller sub programs is called **modularization**. Computer programming languages have different methods to implement modularization. The sub programs are generally called functions. C++ also facilitates modular programming with functions.

3.1.1 Merits of modular programming

The modular style of programming has many advantages. It reduces the complexity and size of the program, makes the program more readable, improves re-usability and makes the debugging process easy. Let us discuss these features in detail:

Reduces the size of the program: In some cases, certain instructions in a program may be repeated at different points of the program. Consider the expression $\frac{x^5 + y^7}{\sqrt{x} + \sqrt{y}}$. To evaluate this expression for given values of x and y, we have to use instructions for the following:

1. find the 5th power of x.
2. find the 7th power of y.
3. add the results obtained in steps 1 and 2.
4. find the square root of x.
5. find the square root of y.
6. add the result obtained in steps 4 and 5.
7. divide the result obtained in step 3 by that in step 6.

We know that separate loops are needed to find the results of steps 1 and 2. Can you imagine the complexity of the logic to find the square root of a number? It is clear that the program requires the same instructions to process different data at different points. The modular approach helps to isolate the repeating task and write instructions for this. We can assign a name to this set of instructions and this can be invoked by using that name. Thus program size is reduced.

Less chance of error occurrence: When the size of program is reduced, naturally syntax errors will be less in number. The chance of logical error will also be minimized because in a modularized program, we need to concentrate only on one module at a time.

Reduces programming complexity: The net result of the two advantages discovered above is reducing programming complexity. If we properly divide the problem into smaller conceptual units, the development of logic for the solution will be simpler.

Improves reusability: A function written once may be used later in many other programs, instead of starting from scratch. This reduces the time taken for program development.

3.1.2 Demerits of modular programming

Though there are significant merits in modular programming, proper breaking down of the problem is a challenging task. Each sub problem must be independent of others. Utmost care should be given while setting the hierarchy of the execution of the modules.

3.2 Functions in C++

Let us consider the case of a coffee making machine and discuss its function based on Figure 3.2. Water, milk, sugar and coffee powder are supplied to the machine. The machine

Fig. 3.2 : Function of coffee making machine

processes it according to a set of predefined instructions stored in it and returns coffee which is collected in a cup. The instruction-set may be as follows:

1. Get 60 ml milk, 120 ml water, 5 gm coffee powder and 20 gm sugar from the storage of the machine.
2. Boil the mixture
3. Pass it to the outlet.

Usually there will be a button in the machine to invoke this procedure. Let us name the button with the word "MakeCoffee". Symbolically we can represent the invocation as:

Cup = MakeCoffee (Water, Milk, Sugar, Coffee Powder)

We can compare all these aspects with functions in programs. The word "MakeCoffee" is the **name** of the function, "Water", "milk", "sugar" and "coffee powder" are **parameters** for the function and "coffee" is the result **returned**. It is **stored** in a "Cup". Instead of cup, we can use a glass, tumbler or any other container.

Similarly, a C++ function accepts parameters, processes it and returns the result. Figure 3.3 can be viewed as a function **Add** that accepts 3, 5, 2 and 6 as parameters, adds them and returns the value which is stored in the variable **C**. It can be written as:

C = Add (3, 5, 2, 6)

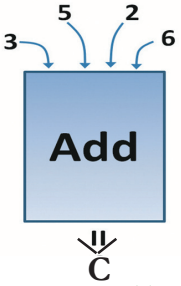

Fig. 3.3. Addition function

We can say that ***function*** is a named unit of statements in a program to perform a specific task as part of the solution. It is not necessary that all the functions require some parameters and all of them return some value. C++ provides a rich collection of functions ready to be used for various tasks. The tasks to be performed by each of these are already written, debugged and compiled, their definitions alone are grouped and stored in files called header files. Such ready-to-use sub programs are called ***predefined functions*** or ***built-in functions***.

While writing large programs, the predefined functions may not suffice to apply modularization. C++ provides the facility to create our own functions for some specific tasks. Everything related to a function such as the task to be carried out, the name and data required are decided by the user and hence they are known as ***user-defined functions***.

What is the role of **`main()`** function then? It may be considered as user-defined in the sense that the task will be defined by the user. We have learnt that it is an essential function in a C++ program because the execution of a program begins in `main()`.

Without `main()`, C++ program will not run. All other functions will be executed when they are called or invoked (or used) in a statement.

3.3 Predefined functions

C++ provides a number of functions for various tasks. We will discuss only the most commonly used functions. While using these functions, some of them require data for performing the task assigned to it. We call them ***parameters*** or ***arguments*** and are provided within the pair of parentheses of the function name. There are certain functions which give results after performing the task. This result is known as ***return-value*** of the function. Some functions do not return any value, rather they perform the specified task. In the following sections, we discuss functions for manipulating strings, performing mathematical operations and processing character data. While using these functions the concerned header files are to be included in the program.

3.3.1 Console functions for character I/O

We have discussed functions for input/output operations on strings. C++ also provides some functions for performing input/ouput operations on characters. In Chapter 2, we discussed `gets()` function and its advantage in string input. Now let us see some functions to input and output character data. These functions require the inclusion of header file **`cstdio`** (stdio.h in Turbo C++ IDE) in the program.

`getchar()`

This function returns the character that is input through the keyboard. The character can be stored in a variable as shown in the example given below:

```
char ch = getchar();
```

In the previous chapter we have seen `puts()` function and its advantage in string output. Let us have a look at a function used to output character data.

`putchar()`

This function displays the character given as the argument on the standard output unit (monitor). The argument may be a character constant or a variable. If an integer value is given as the argument, it will be considered as an ASCII value and the corresponding character will be displayed. The following code segment illustrates the use of `putchar()` function.

```
char ch = 'B';      //assigns 'B' to the variable ch
putchar(ch);        //displays 'B' on the screen
putchar('c');       //displays 'c' on the screen
putchar(97);        //displays 'a' on the screen
```

Program 3.1 illustrates the working of these functions. This program allows inputting a string and a character to be searched.

Program 3.1: To search for a character in a string using console functions

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    char str[20], ch;
    int i, num=0;
    puts("Enter a string:"); //To print '\n' after the string
    gets(str);//To accept a string with white spaces
    cout<<"\nEnter the character to be searched: ";
    ch=getchar();//To input the character to be searched
    /* A loop to search for the character and count its
       occurrences in the string. Search will be
       terminated when null character is found   */
    for(i=0; str[i]!='\0'; i++)
       if (str[i]==ch)
          num++;
    cout<<"\nThe number of occurences of the character ";
    putchar(ch);
    cout<<" is : "<<num;
    return 0;
}
```

Sample output of the above program is given below

```
Enter a string:
examination
Enter the character to be searched: a
The number of occurences of the character a is : 2
```

3.3.2 Stream functions for I/O operations

C++ provides another facility to perform input/output operations on character and strings. It is in the form of functions that are available in the header file **iostream**. These functions are generally called stream functions since they allow a stream of bytes (data) to flow between memory and objects. Devices like the keyboard and the monitor are referenced as objects in C++. Let us discuss some of these functions.

A. Input functions

These functions allow the input of character and string data. The input functions such as **get()** and **getline()** allow a stream of bytes to flow from input object into the memory. The object **cin** is used to refer to keyboard and hence whenever we input data using keyboard, these functions are called or invoked using this object as **cin.get()** and **cin.getline()**. Note that a period symbol (.), called ***dot operator*** is used between the object `cin` and the function.

i. get()

It can accept a single character or multiple characters (string) through the keyboard. To accept a string, an array name and size are to be given as arguments. Following code segment illustrates the usage of this function.

```
char ch, str[10];
ch=cin.get(ch); //accepts a character and stores in ch
cin.get(ch);    //equivalent to the above statement
cin.get(str,10); //accepts a string of maximum 10 characters
```

ii. getline()

It accepts a string through the keyboard. The delimiter will be Enter key, the number of characters or a specified character. This function uses two syntaxes as shown in the code segment given below.

```
char ch, str[10];
int len;
cin.getline(str,len);        // With 2 arguments
cin.getline(str,len,ch);     // With 3 arguments
```

In the first usage, `getline()` function has two arguments - a character array (here it is, `str`) and an integer (`len`) that represents maximum number of characters that can be stored. In the second usage, a delimiting character (content of `ch`) can also be given along with the number of characters. While inputting the string, only (`len-1`) characters, or characters upto the specified delimiting character, whichever occurs first will be stored in the array.

B. Output functions

Output functions like **put()** and **write()** allow a stream of bytes to flow from memory into an output object. The object **cout** is used with these functions since we use the monitor for the output.

i. put()

It is used to display a character constant or the content of a character variable given as argument.

```
char ch='c';
cout.put(ch);      //character 'c' is displayed
cout.put('B');     //character 'B' is printed
cout.put(65);      //character 'A' is printed
```

ii. write()

This function displays the string contained in the argument. For illustration see the example given below.

```
char str[10]="hello";
cout.write(str,10);
```

The above code segment will dispaly the string `hello` followed by 5 white spaces, since the second argument is 10 and the number of characters in the string is 5.

Program 3.2: To illustrate the working of stream input/output functions

```
#include <iostream>
#include <cstring> //To use strlen() function
using namespace std;
int main()
{
    char ch, str[20];
    cout<<"Enter a character: ";
    cin.get(ch); //To input a character to the variable ch
    cout<<"Enter a string: ";
    cin.getline(str,10,'.');     //To input the string
    cout<<"Entered character is:\t";
    cout.put(ch);           //To display the character
    cout.write("\nEntered string is:",20);
    cout.write(str,strlen(str));
    return 0;
}
```

On executing Program 3.2, the following output will be obtained:

```
Enter a character: p
Enter a string: hello world
Entered character is:     p
Entered string is:
hello wo
```

Let us discuss what happens when the program is executed. In the beginning, `get()` function allows to input a character, say `p`. When the function `getline()` is executed, we can input a string, say `hello world`. The `put()` function is then executed to

display the character p. Observe that the write() function displays only hello wo in a new line. In the getline() function, we specified the integer 10 as the maximum number of characters to be stored in the array str. Usually 9 characters will be stored, since one byte is reserved for '\0' character as the string terminator. But the output shows only 8 characters including white space. This is because, the Enter key followed by the character input (p) for the get() function, is stored as the '\n' character in the first location of str. That is why, the string, hello wo is displayed in a new line.

If we run the program, by giving the input hello.world, the output will be as follows: Observe the change in the content of str.

```
Enter a character: a
Enter a string: hello.world
Entered character is:      a
Entered string is:
hello
```

The change has occurred because the getline() function accepts only the characters that appear before the dot symbol.

3.3.3 String functions

While solving problems string data may be involved for processing. C++ provides string functions for their manipulation. The header file **cstring** (string.h in Turbo C++) is to be included in the program to use these functions.

i. strlen()

This function is used to find the length of a string. Length of a string means the number of characters in the string. Its syntax is:

```
int strlen(string);
```

This function takes a string as the argument and gives the length of the string as the result. The following code segment illustrates this.

```
char str[] = "Welcome";
int n;
n = strlen(str);
cout << n;
```

Here, the argument for the strlen() function is a string variable and it returns the number of characters in the string, i.e. 7 to the variable n. Hence the program code will display 7 as the value of the variable n. The output will be the same even though the array declaration is as follows.

```
char str[10] = "Welcome";
```

Note that the array size is specified in the declaration. The argument may be a string constant as shown below:

```
n = strlen("Computer");
```

The above statement returns 8 and will be stored in n.

ii. strcpy()

This function is used to copy one string into another. The syntax of the function is:

```
strcpy(string1, string2);
```

The function will copy string2 to string1. Here string1 is an array of characters and string2 is an array of characters or a string constants. These are the arguments for the execution of the function. The following code illustrates its working:

```
char s1[10], s2[10] = "Welcome";
strcpy(s1,s2);
cout << s1;
```

The string "Welcome" contained in the string variable s1 will be displayed on the screen. The second argument may be a string constant as follows:

```
char str[10]
strcpy(str,"Welcome");
```

Here, the string constant "Welcome" will be stored in the variable str. The assignment statement, str = "Welcome"; is wrong. But we can directly assign value to a character array at the time of declaration as:

```
char str[10] = "Welcome";
```

iii. strcat()

This function is used to append one string to another string. The length of the resultant string is the total length of the two strings. The syntax of the functions is:

```
strcat(string1, string2);
```

Here string1 and string2 are array of characters or string constants. string2 is appended to string1. So, the size of the first argument should be able to accommodate both the strings together. Let us see an example showing the usage of this function:

```
char s1[20] = "Welcome", s2[10] = " to C++";
strcat(s1,s2);
cout << s1;
```

The above program code will display "Welcome to C++" as the value of the variable s1. Note that the string in s2 begins with a white space.

iv. strcmp()

This function is used to compare two strings. In this comparison, the alphabetical order of characters in the strings is considered. The syntax of the function is:

```
strcmp(string1, string2)
```

The function returns any of the following values in three different situations.

- Returns 0 if `string1` and `string2` are same.
- Returns a –ve value if `string1` is alphabetically lower than `string2`.
- Returns a +ve value if `string1` is alphabetically higher than `string2`.

The following code fragment shows the working of this function.

```
char s1[]="Deepthi", s2[]="Divya";
int n;
n = strcmp(s1,s2);
if(n==0)
    cout<<"Both the strings are same";
else if(n < 0)
    cout<<"s1 < s2";
else
    cout<<"s1 > s2";
```

It is clear that the above program code will display `"s1 < s2"` as the output.

v. strcmpi()

This function is used to compare two strings ignoring cases. That is, the function will treat both the upper case and lower case letters as same for comparison. The syntax and working of the function are the same as that of `strcmp()` except that `strcmpi()` is not case sensitive. This function also returns values as in the case of `strcmp()`. Consider the following code segment:

```
char s1[]="SANIL", s2[]="sanil";
int n;
n = strcmpi(s1,s2);
if(n==0)
    cout<<"strings are same";
else if(n < 0)
    cout<<"s1 < s2";
else
    cout<<"s1 > s2";
```

The above program code will display `"strings are same"` as the output, because the uppercase and lowercase letters will be treated as same during the comparison.

Program 3.3 compares and concatenates two strings. The length of the newly formed string is also displayed.

Program 3.3 : To combine two strings if they are different and find its length

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char s1[15], s2[15], s3[30];
    cout<<"Enter two strings: ";
    cin>>s1>>s2;
    int n;
    n=strcmp(s1,s2);
    if (n==0)
        cout<<"\nThe input strings are same";
    else
    {
        cout<<"\nThe input strings are not same";
        strcpy(s3,s1);//Copies the string in s1 into s3
        strcat(s3,s2);//Appends the string in s2 to that in s3
        cout<<"String after concatenation is: "<<s3;
        cout<<"\nLength of the new string is: "
             <<strlen(s3);
    }
    return 0;
}
```

Header file essential for using string manipulating functions

Sample Output:

```
Enter two strings:india
kerala
The input strings are not same
String after concatenation is:indiakerala
Length of the new string is: 11
```

3.3.4 Mathematical functions

Now, let us discuss the commonly used mathematical functions available in C++. We should include the header file **cmath** (math.h in Turbo C++) to use these functions in the program.

i. abs()

It is used to find the absolute value of an integer. It takes an integer as the argument (+ve or –ve) and returns the absolute value. Its syntax is:

```
int abs(int)
```

The following is an example to show the output of this function:

```
int n = -25;
cout << abs(n);
```

The above program code will display 25. If we want to find the absolute value of a floating point number, we can use **fabs()** function as used above. It will return the floating point value.

ii. sqrt()

It is used to find the square root of a number. The argument to this function can be of type `int`, `float` or `double`. The function returns the non-negative square root of the argument. Its syntax is:

```
double sqrt(double)
```

The following code snippet is an example. This code will display 5.

```
int n = 25;
float b = sqrt(n);
cout << b;
```

If the value of n is 25.4, then the answer will be 5.03841

iii. pow()

This function is used to find the power of a number. It takes two arguments **x** and **y**. The argument **x** and **y** are of type `int`, `float` or `double`. The function returns the value of $\mathbf{x}^{\mathbf{y}}$. Its syntax is:

```
double pow(double, double)
```

The following example shows the working of this function.

```
int x = 5, y = 4, z;
z = pow(x, y);
cout << z;
```

The above program code will display 625.

x^y means 5^4 which is 5*5*5*5 i.e. 625

Program 3.4 : To find the area of a triangle and a circle using mathematical functions

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    const float pi=22.0/7;
    int a,b,c, radius;
    float s, area1, area2;
    cout<<"Enter the three sides of the triangle: ";
    cin>>a>>b>>c;
    s = (a+b+c)/2.0;
    area1 = sqrt(s*(s-a)*(s-b)*(s-c));
    cout<<"The Area of the Triangle is: "<<area1;
    cout<<"\nEnter the radius of the circle: ";
    cin>>radius;
    area2 = pi*pow(radius,2);
    cout<<"Area of the Circle is: "<<area2;
    return 0;
}
```

Header file essential for using mathematical functions

The following is a sample output of the above program:

```
Enter the three sides of the triangle: 5    7    9
The Area of the Triangle is: 17.4123
Enter the radius of the circle: 2.5
Area of the Circle is: 12.5714
```

3.3.5 Character functions

These functions are used to perform various operations on characters. Following are the various character functions available in C++. The header file **cctype** (ctype.h in Turbo C++) is to be included to use these functions in a program.

i. isupper()

This function is used to check whether a character is in upper case (capital letter) or not. The syntax of the function is:

```
int isupper(char c)
```

The function returns 1 if the given character is in uppercase, and 0 otherwise.

The following statement assigns 0 to the variable n.

```
int n = isupper('x');
```

Consider the following statements:

```
char c = 'A';
int n = isupper(c);
```

The value of the variable n, after the execution of the above statements will be 1, since the given character is in upper case.

ii. islower()

This function is used to check whether a character is in lower case (small letter) or not. The syntax of the function is:

```
int islower(char c)
```

The function returns 1 if the given character is lower case, and 0 otherwise.

After executing the following statements, the value of the variable n will be 1 since the given character is in lower case.

```
char ch = 'x';
int n = islower(ch);
```

But the statement given below assigns 0 to the variable n, since the given character is in the uppercase.

```
int n = islower('A');
```

iii. isalpha()

This function is used to check whether the given character is an alphabet or not. The syntax of the function is:

```
int isalpha(char c)
```

The function returns 1 if the given character is an alphabet, and 0 otherwise.

The following statement assigns 0 to the variable n, since the given character is not an alphabet.

```
int n = isalpha('3');
```

But the statement given below displays 1, since the given character is an alphabet.

```
cout << isalpha('a');
```

iv. isdigit()

This function is used to check whether the given character is a digit or not. The syntax of the function is:

```
int isdigit(char c)
```

The function returns 1 if the given character is a digit, and 0 otherwise.

After executing the following statement, the value of the variable n will be 1 since the given character is a digit.

```
n = isdigit('3');
```

When the following statements are executed, the value of the variable n will be 0, since the given character is not a digit.

```
char c = 'b';
int n = isdigit(c);
```

v. isalnum()

This function is used to check whether a character is alphanumeric or not. The syntax of the function is:

```
int isalnum (char c)
```

The function returns 1 if the given character is alphanumeric, and 0 otherwise.

Each of the following statements returns 1 after the execution.

```
n = isalnum('3');
cout << isalnum('A');
```

But the statements given below assigns 0 to the variable n, since the given character is neither alphabet nor digit.

```
char c = '-';
int n = isalnum(c);
```

vi. toupper()

This function is used to convert the given character into its uppercase. The syntax of the function is:

```
char toupper(char c)
```

The function returns the upper case of the given character. If the given character is in upper case, the output will be the same.

The following statement assigns the character constant 'A' to the variable c.

```
char c = toupper('a');
```

But the output of the statement given below will be 'A' itself.

```
cout << (char)toupper('A');
```

Note that type conversion using (char) is used in this statement. If conversion method is not used, the output will be 65, which is the ASCII code of 'A'.

vii. tolower()

This function is used to convert the given character into its lower case. The syntax of the function is:

```
char tolower(char c)
```

The function returns the lower case of the given character. If the given character is in lowercase, the output will be the same.

Consider the statement: `c = tolower('A');`

After executing the above statement, the value of the variable `c` will be `'a'`. But when the following statements will be executed, the value of the variable `c` will be `'a'` itself.

```
char x = 'a';
char c = tolower(x) ;
```

In the case of functions `tolower()` and `toupper()`, if the argument is other than alphabet, the given character itself will be returned on execution.

Program 3.5 illustrates the use of character functions. This program accepts a line of text and counts the lowercase letters, uppercase letters and digits in the string. It also displays the entire string both in uppercase and lowercase.

Program 3.5: To count different types of characters in the given string

```
#include <iostream>
#include <cstdio>
#include <cctype>
using namespace std;
int main()
{
    char text[80];
    int Ucase=0, Lcase=0, Digit=0;
    cout << "Enter a line of text: ";
    gets(text);
    for(int i=0; text[i]!='\0'; i++)
        if (isupper(text[i]))
            Ucase++;
        else if (islower(text[i]))
            Lcase++;
        else if (isdigit(text[i]))
            Digit++;
```

Loop will be terminated when the value of `i` point to the null character

```
    cout << "\nNo. of uppercase letters = " << Ucase;
    cout << "\nNo. of lowercase letters = " << Lcase;
    cout << "\nNo. of digits = " << Digit;
    cout << "\nThe string in uppercase form is\n";
    i=0;
    while (text[i]!='\0')
    {
        putchar(toupper(text[i]));
        i++;
    }
    cout << "\nThe string in lowercase form is\n";
    i=0;
    do
    {
        putchar(tolower(text[i]));
        i++;
    }while(text[i]!='\0');
    return 0;
}
```

If `cout<<` is used instead of `putchar()`, the ASCII code of the characters will be displayed

A sample output is given below:

```
Enter a line of text : The vehicle ID is KL01 AB101
No. of uppercase letters = 7
No. of lowercase letters = 11
No. of digits = 5
The string in uppercase form is
THE VEHICLE ID IS KL01 AB101
The string in lowercase form is
the vehicle id is kl01 ab101
```

Let us do

Prepare a chart in the following format and fill up the columns with all predefined functions we have discussed so far.

Function	Usage	Syntax	Example	Output

Know your progress

1. What is modular programming?
2. What is meant by a function in C++?
3. Name the header file required for using character functions.
4. Predict the output of `cout<<sqrt(49)`.
5. Pick the odd one out and give reason:
   a. `strlen()` b. `pow()` c. `strcpy()` d. `strcat( )`
6. Name the header file needed for the function pow().
7. Predict the output when we compare the strings "HELLO" and "hello" using the function `strcmpi()`.
8. What will be output of the statement
   `cout<<strlen("smoking kills");` ?
9. Which function converts the alphabet 'P' to 'p'?
10. Identify the name of the function which tests whether the argument is an aphabet or number.

3.4 User-defined functions

All the programs that we have discussed so far contain a function named **`main()`**. We know that a C++ program begins with preprocessor directive statements followed by `using namespace` statement. The remaining part is actually the definition of a function. The `int main()` in the programs is called ***function header*** (or function heading) of the function and the statements within the pair of braces immediately after the header is called its ***body***.

The syntax of a function definition is given below:

```
data_type  function_name(argument_list)
{
     statements in the body;
}
```

The `data_type` is any valid data type of C++. The `function_name` is a user-defined word (identifier). The `argument_list`, which is optional, is a list of parameters, i.e. a list of variables preceded by data types and separated by commas. The body comprises of C++ statements required to perform the task assigned to the function. Once we have decided to create a function, we have to answer certain questions.

(i) Which data type will be used in the function header?

(ii) How many arguments are required and what should be the preceding data type of each?

Let us recollect how we have used the predefined functions `strcpy()` and `sqrt()`. We have seen that these functions will be executed when they are called (or used) in a C++ statement. The function `getchar()` takes no arguments in the parentheses. But for `strcpy()`, two strings are provided as arguments or parameters. Without these arguments this function will not work, because it is defined with two string (character array) arguments. In the case of `sqrt()`, it requires a numeric data as argument and gives a result (of `double` type) after performing the predefined operations on the given argument. This result, as mentioned earlier, is called the **return-value** of the function. The data type of a function depends on this value. In other words we can say that the function should return a value which is of the same data type of the function. So, the data type of a function is also known as the **return type** of the function. Note that we use `return 0;` statement in `main()` function, since it is defined with `int` data type as per the requirement of GCC.

The number and type of arguments depend upon the data required by the function for processing. Note that a function can return only one value. But some functions like `puts()` and `gets()` do not return any value. The header of such functions uses `void` as the return type. A function either returns one value or nothing at all.

3.4.1 Creating user-defined functions

Based on the syntax discussed above, let us create some functions. The following is a function to display a message.

```
void saywelcome()
{
    cout<<"Welcome to the world of functions";
}
```

The name of the function is `saywelcome()`, its data type (return type) is `void` and it does not have any argument. The body contains only one statement.

Now, let us define a function to find the sum of two numbers. Four different types of definitions are given for the same task, but they vary in definition style and hence the usage of each is different from others.

Function 1	*Function 2*

`void sum1()`
`{`
`  int a, b, s;`
`  cout<<"Enter 2 numbers: ";`
`  cin>>a>>b;`
`  s=a+b;`
`  cout<<"Sum="<<s;`
`}`
	`int sum2()`
	`{`
	`  int a, b, s;`
	`  cout<<"Enter 2 numbers: ";`
	`  cin>>a>>b;`
	`  s=a+b;`
	`  return s;`
	`}`

Function 3	*Function 4*

`void sum3(int a, int b)`
`{`
`  int s;`
`  s=a+b;`
`  cout<<"Sum="<<s;`
`}`
	`int sum4(int a, int b)`
	`{`
	`  int s;`
	`  s=a+b;`
	`  return s;`
	`}`

Let us analyse these functions and see how they are different. The task is the same in all these functions, but they differ in the number of parameters and return type.

Table 3.1 shows that the function defined with a data type other than **`void`** should return a value in accordance with the data type. The **`return`** statement is used for this purpose (Refer to functions 2 and 4). The **`return`** statement returns a value to the calling function and transfers the program control back to the calling function. So, remember that if a `return` statement is executed in a function, the remaining statements within that function will not be executed.

Name	Arguments	Return value
sum1()	No arguments	Does not return any value
sum2()	No arguments	Returns an integer value
sum3()	Two integer arguments	Does not return any value
sum4()	Two integer arguments	Returns an integer value

Table 3.1 : Analysis of functions

In most of the functions, `return` is placed at the end of the function. The functions defined with void data type may not have a `return` statement within the body. But if we use `return` statement, we cannot provide any value to it.

Now, let us see how these functions are to be called and how they are executed. We know that no function other than `main()` is executed automatically. The sub functions, either predefined or user-defined, will be executed only when they are called from `main()` function or other user-defined function. The code segments within the rectangles of the following program shows the function calls. Here the `main()` is the calling function and `sum1()`, `sum2()`, `sum3()`, and `sum4()` are the called functions.

```
int main()
{
    int x, y, z=5, result;
    cout << "\nCalling the first function\n";
    sum1();
    cout << "\nCalling the second function\n";
    result = sum2();
    cout << "Sum given by function 2 is " << result;
    cout << "\nEnter values for x and y : ";
    cin >> x >> y;
    cout << "\nCalling the third function\n";
    sum3(x,y);
    cout << "\nCalling the fourth function\n";
    result = sum4(z,12);
    cout << "Sum given by function 4 is " << result;
    cout << "\nEnd of main function";
    return 0;
}
```

The output of the program will be as follows:

```
Calling the first function
Enter 2 numbers: 10  25
Sum=35
Calling the second function
Enter 2 numbers: 5   7
Sum given by function 2 is 12
Enter values for x and y : 8   13
Calling the third function
Sum=21
Calling the fourth function
Sum given by function 4 is 17
End of main function
```

Input for a and b of sum1()

Input for a and b of sum2()

Input for x and y of main()

Function 4 requires two numbers for the task assigned and hence we provide two arguments. The function performs some calculations and gives a result. As there is only one result, it can be returned. Comparatively this function is a better option to find the sum of any two numbers.

Now, let us write a complete C++ program to find the product of two numbers. We will write the program using a user-defined function. But where do we write the

user-defined function in a C++ program? The following table shows two styles to place the user-defined function:

Program 3.6 **- Product of two numbers -** **Program 3.7**

Function before main()

```
#include <iostream>
using namespace std;
int product(int a, int b)
{
   int p;
   p = a * b;
   return p;
}
//Definition above main()
int main()
{
   int ans, num1,num2;
   cout<<"Enter 2 Numbers:";
   cin>>num1>>num2;
   ans = product(num1,num2);
   cout<<"Product = "<<ans;
   return 0;
}
```

Function after main()

```
#include <iostream>
using namespace std;
int main()
{
   int ans, num1,num2;
   cout<<"Enter 2 Numbers:";
   cin>>num1>>num2;
   ans = product(num1,num2);
   cout<<"Product = "<<ans;
   return 0;
}
//Definition below main()
int product(int a, int b)
{
   int p;
   p = a * b;
   return p;
}
```

When we compile Program 3.6, there will be no error. But if we compile Program 3.7, there will be an error `'product was not declared in this scope'`. Let us see what this error means.

3.4.2 Prototype of functions

We have seen that a C++ program can contain any number of functions. But it must have a `main()` function to begin the execution. We can write the definitions of functions in any order as we wish. We can define the `main()` function first and all other functions after that or vice versa. Program 3.6 contains the `main()` function after the user-defined function, but in Program 3.7, the `main()` is defined before the user-defined function. When we compile this program, it will give an error - "`product was not declared in this scope`". This is because the function `product()` is called in the program, before it is defined. During the compilation of the `main()` function, when the compiler encounters the function call `product()`, it is not aware of such a function. Compiler is unable to check whether there is such a function, whether its usage is correct or not, and whether it is accessible or not. So

it reports an error. This error can be removed by giving a declaration statement about the function and this statement is known as prototype. A **function prototype** is the declaration of a function by which compiler is provided with the information about the function such as the name of the function, its return type, the number and type of arguments, and its accessibility. This information is essential for the compiler to verify the correctness of the function call in the program. This information is available in the function header and hence the header alone will be written as a statement before the function call. The following is the format:

```
data_type  function_name(argument_list);
```

In the prototype, the argument names need not be specified. Number and type of arguments must be specified.

So, the error in Program 3.7 can be rectified by inserting the following statement before the function call in the `main()` function.

```
int product(int, int);
```

Like a variable declaration, a function must be declared before it is used in the program. If a function is defined before it is used in the program, there is no need to declare the function separately. The declaration statement may be given outside the `main()` function. The position of the prototype differs in the accessibility of the function. We will discuss this later in this chapter. Wherever be the position of the function definition, execution of the program begins in `main()`.

3.4.3 Arguments of functions

We have seen that functions have arguments or parameters for getting data for processing. Let us see the role of arguments in function call. Consider the function given below:

```
float SimpleInterest(int P, int N, float R)
{
    float amt;
    amt = P * N * R / 100;
    return amt;
}
```

This function gives the simple interest of a given principal amount for a given period at a given rate of interest. The following code segment illustrates different function calls:

```
cout << SimpleInterest(1000,3,2);//Function call 1
int x, y; float z=3.5, a;
cin >> x >> y;
a = SimpleInterest(x, y, z); //Function call 2
```

When the first statement is executed, the values 1000, 3 and 2 are passed to the argument list in the function definition. The arguments `P`, `N` and `R` get the values 1000, 3 and 2, respectively. Similarly, when the last statement is executed, the values of the variables `x`, `y` and `z` are passed to the arguments `P`, `N` and `R`.

The variables `x`, `y` and `z` are called actual (original) arguments or actual parameters since they are the actual data passed to the function for processing. The variables `P`, `N` and `R` used in the function header are known as formal arguments or formal parameters. These arguments are intended to receive data from the calling function. **Arguments or parameters** are the means to pass values from the calling function to the called function. The variables used in the function definition as arguments are known as **formal arguments**. The constants, variables or expressions used in the function call are known as **actual (original) arguments**. If variables are used in function prototype, they are known as dummy arguments.

Now, let us write a program that uses a function `fact()` that returns the factorial of a given number to find the value of nCr. As we know, factorial of a number N is the product of the first N natural numbers. The value of nCr is calculated by the formula $\frac{n!}{r!(n-r)!}$, where n! denotes the factorial of n.

Program 3.8: To find the value of nCr

```
#include<iostream>
using namespace std;
int fact(int);   //Function prototype
int main()
{
    int n,r, ncr;
    cout<<"Enter the values of n and r : ";
    cin>>n>>r;
    ncr=fact(n)/(fact(r)*fact(n-r));
    cout<<n<<"C"<<r<<" = "<<ncr;
    return 0;
}
int fact(int N)  //Function header
{
    int f;
    for(f=1; N>0; N--)
        f=f*N;
    return f;
}
```

Actual arguments

Function call according to the formula

Formal argument

Factorial is returned

The following is a sample output:

```
Enter the values of n and r : 5     2
5C2 = 10
```

3.4.4 Functions with default arguments

Let us consider a function `TimeSec()` with the argument list as follows. This function accepts three numbers that represent time in hours, minutes and seconds. The function converts this into seconds.

```
int TimeSec(int H, int M=0, int S=0)
{
    int sec = H * 3600 + M * 60 + S;
    return sec;
}
```

Note that the two arguments `M` and `S` are given default value 0. So, this function can be invoked in the following ways.

```
long s1 = TimeSec(2,10,40);
long s2 = TimeSec(1,30);
long s3 = TimeSec(3);
```

It is important to note that all the default arguments must be placed from the right to the left in the argument list. When a function is called, actual arguments are passed to the formal arguments from left onwards.

When the first statement is executed, the function is called by passing the values 2, 10 and 40 to the formal parameters `H`, `M` and `S`, respectively. The initial values of `M` and `S` are over-written by the actual arguments. During the function call in the second statement, `H` and `M` get values from actual arguments, but `S` works with its default value 0. Similarly, when the third statement is executed, `H` gets the value from calling function, but `M` and `S` use the default values. So, after the function calls, the values of `s1`, `s2` and `s3` will be 7840, 5400 and 10800, respectively.

We have seen that functions can be defined with arguments assigned with initial values. The initialized formal arguments are called **default arguments** which allow the programmer to call a function with different number of arguments. That is, we can call the function with or without giving values to the default arguments.

3.4.5 Methods of calling functions

Suppose your teacher asks you to prepare invitation letters for the parents of all students in your class, requesting them to attend a function in your school. The teacher can give you a blank format of the invitation letter and also a list containing

the names of all the parents. The teacher can give you the name list in two ways. She can take a photo copy of the name list and give it to you. Otherwise, she can give the original name list itself. What difference would you feel in getting the name list in these two ways? If the teacher gives the original name list, you will be careful not to make any marks or writing in the name list because the teacher may want the same name list for future use. But if you are given a photo copy of the name list, you can make any marking in the list, because the change will not affect the original name list.

Let us consider the task of preparing the invitation letter as a function. The name list is an argument for the function. The argument can be passed to the function in two ways. One is to pass a copy of the name list and the other is to pass the original name list. If the original name list is passed, the changes made in the name list, while preparing the invitation letter, will affect the original name list. Similarly, in C++, an argument can be passed to a function in two ways. Based on the method of passing the arguments, the function calling methods can be classified as Call by Value method and Call by Reference method. The following section describes the methods of argument passing in detail.

a. Call by value (Pass by value) method

In this method, the value contained in the actual argument is passed to the formal argument. In other words, a copy of the actual argument is passed to the function. Hence, if the formal argument is modified within the function, the change is not reflected in the actual argument at the calling place. In all previous functions, we passed the argument by value only. See the following example:

```
void change(int n)
{
    n = n + 1;
    cout << "n = " << n << '\n';
}
void main()
{
    int x = 20;
    change(x);
    cout << "x = " << x;
}
```

The parameter n has its own memory location to store the value 20 in change().

The value of x is passed to n in change()

When we pass an argument as specified in the above program segement, only a copy of the variable x is passed to the function. In other words, we can say that only the value of the variable x is passed to the function. Thus the formal parameter n in

the function will get the value 20. When we increase the value of n, it will not affect the value of the variable x. The following will be the output of the above code:

```
n = 21
x = 20
```

Table 3.2 shows what happens to the arguments when a function is called using call-by-value method:

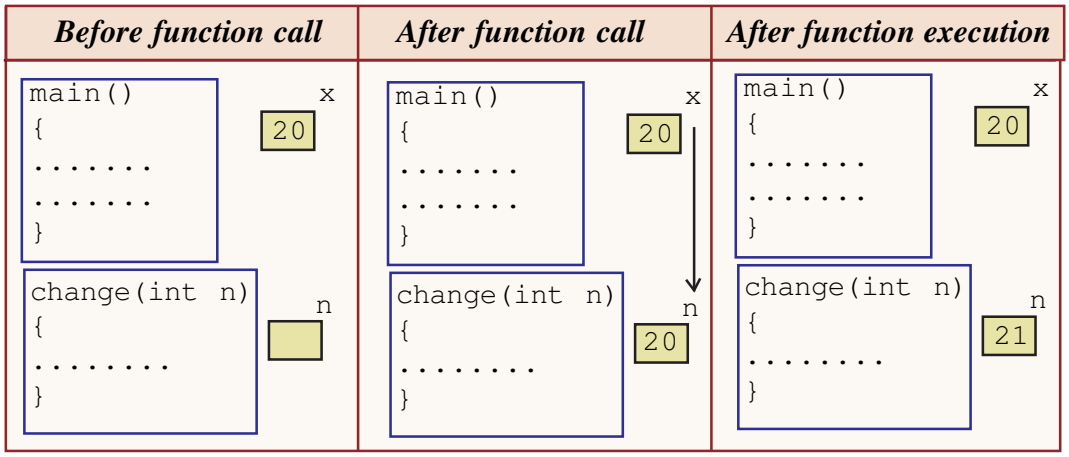

Table 3.2: Call by value procedure

b. Call by reference (Pass by reference) method

When an argument is passed by reference, the reference of the actual argument is passed to the function. As a result, the memory location allocated to the actual argument will be shared by the formal argument. So, if the formal argument is modified within the function, the change will be reflected in the actual argument at the calling place. In C++, to pass an argument by reference we use reference variable as formal parameter. A **reference variable** is an alias name of another variable. An ampersand symbol (**&**) is placed in between the data type and the variable in the function header. Reference variables will not be allocated memory exclusively like the other variables. Instead, it will share the memory allocated to the actual arguments. The following function uses reference variable as formal parameter and hence call by reference method is implemented for function call.

```
void change(int & n)
{
    n = n + 1;
    cout << "n = " << n << '\n';
}
```

The parameter n is a reference variable and hence there is no exclusive memory allocation for it

```
void main()
{
    int x=20;
    change(x);
    cout << "x = " << x;
}
```

The reference of x will be passed to n of the change() function, which results into the sharing of memory.

Note that the only change in the `change()` function is in the function header. The **&** symbol in the declaration of the parameter `n` means that the argument is a reference variable and hence the function will be called by passing reference. Hence when the argument `x` is passed to the `change()` function, the variable `n` gets the address of `x` so that the location will be shared. In other words, the variables `x` and `n` refer to the same memory location. We use the name `x` in the `main()` function, and the name `n` in the `change()` function to refer the same storage location. So, when we change the value of `n`, we are actually changing the value of `x`. If we run the above program, we will get the following output:

```
n = 21
x = 21
```

Table 3.3 depicts the changes in the arguments when call-by-reference is applied for the function call.

Before function call	*After function call*	*After function execution*
main() { .... } x 20 change(int &n) { ..... }	main() { .... } x 20 n change(int &n) { ..... }	main() { .... } x 21 n change(int &n) { ..... }

Table 3.3: Call by value reference procedure

We have discussed the difference between the two methods of function calling. Basically the methods differ in the mode of passing arguments. Let us consolidate the difference between these two methods of function calling. Table 3.4 gives the clear picture.

Call by Value Method	Call by Reference Method
• Ordinary variables are used as formal parameters. • Actual parameters may be constants, variables or expressions. • The changes made in the formal arguments do not reflect in actual arguments. • Exclusive memory allocation is required for the formal arguments.	• Reference variables are used as formal parameters. • Actual parameters will be variables only. • The changes made in the formal arguments do reflect in actual arguments. • Memory of actual arguments is shared by formal arguments.

Table 3.4 : Call by value v/s Call by reference

Let us discuss a typical example for call by reference method. This program uses a function that can be called by reference method for exchanging the values of the two variables in the `main()` function. The process of exchanging values of two variables is known as swapping.

Program 3.9: To swap the values of two variables

```
#include <iostream>
using namespace std;
void swap(int & x, int & y)
{
    int t = x;
    x = y;
    y = t;
}
int main()
{
    int m, n;
    m = 10;
    n = 20;
    cout<<"Before swapping m= "<< m <<" and n= "<<n;
    swap(m, n);
    cout<<"\nAfter swapping m= "<< m <<" and n= "<<n;
    return 0;
}
```

Let us go through the statements in Program 3.9. The actual arguments `m` and `n` are passed to the function by reference. Within the `swap()` function, the values of

x and y are interchanged. When the values of x and y are changed, actually the change takes place in m and n. Therefore the output of the above program code is:

```
Before swapping m= 10 and n= 20
After swapping m= 20 and n= 10
```

Modify the above program by replacing the formal arguments with ordinary variables and predict the output. Check your answer by executing the code in the lab.

Know your progress

1. Identify the most essential function in C++ programs.
2. List the three elements of a function header.
3. What is function prototype?
4. Which component is used for data transfer from calling function to called function?
5. What are the two parameter passing techniques used in C++?
6. Identify the name of the function call where & symbol is used along with formal arguments.

3.5 Scope and life of variables and functions

We have discussed C++ programs consisting of more than one function. Predefined functions are used by including the header file concerned. User-defined functions are placed before or after the `main()` function. We have seen the relevance of function prototypes while defining functions. We have also used variables in the function body and as arguments. Now, let us discuss the availability or accessibility of the variables and functions throughout the program. Program 3.10 illustrates the accessibility of local variables in a program.

Program 3.10: To illustrate the scope and life of variables

```
#include <iostream>
using namespace std;
int cube(int n)
{
    int cb;
    cout<< "The value of x passed to n is " << x;
    cb = n * n * n;
    return cb;
}
```

This is an error because the variable x is declared within the `main()` function. So it cannot be used in other functions.

```
int main()
{
    int x, result;
    cout << "Enter a number : ";
    cin >> x;
    result = cube(x);
    cout << "Cube = " << result;
    cout << "\nCube = "<< cb;
    return 0;
}
```

This is an error because the variable cb is declared within the cube() function. So it cannot be used in other functions.

When we compile the program, there will be two errors because of the reasons shown in the call-outs. The concept of availability or accessibility of variables and functions is termed as their scope and life time. **Scope** of a variable is that part of the program in which it is used. In the above program, scope of the variable cb is in the cube() function because it is declared within that function. Hence this variable cannot be used outside the function. This scope is known as **local scope**. On completing the execution of a function, memory allocated for all the variables within a function is freed. In other words, we can say that the life of a variable, declared within a function, ends with the execution of the last instruction of the function. So, if we use a variable n within the main(), that will be different from the argument n of a called function or a variable n within the called function. The variables used as formal arguments and/or declared within a function have local scope.

Just like variables, functions also have scope. A function can be used within the function where it is declared. That is the function is said to have local scope. If it is declared before the main() function and not within any other function, the scope of the function is the entire program. That is the function can be used at any place in the program. This scope is known as **global scope**. Variables can also be declared with global scope. Such declarations will be outside all the functions in the program.

Look at Program 3.11 to get more clarity on the scope and life of variables and functions throughout the program.

Program 3.11 : To illustrate the scope and life of variables and functions

```
#include <iostream>
using namespace std;
int cb;   //global variable
void test()//global function since defined above other functions
{
    int cube(int n);  //It is a local function
    cb=cube(x); //Invalid call. x is local to main()
    cout<<cb;
}

int main() // beginning of main() function
{
    int x=5;  //local variable
    test();   //valid call since test() is a global function
    cb=cube(x);  //Invalid call. cube() is local to test()
    cout<<cb;
}
int cube(int n)//Argument n is local variable
{
    int val= n*n*n; //val is local variable
    return val;
}
```

The given comments explain the scope and life of functions. A function which is declared inside the function body of another function is called a ***local function*** as it can be used within that function only. A function declared outside the function body of any other function is called a ***global function***. A global function can be used throughout the program. In other words, the scope of a global function is the entire program and that of a local function is only the function where it is declared. Table 3.5 summarises the scope and life time of variables and functions.

Scope & life	Local	Global
Variables	• Declared within a function or a block of statements. • Available only within that function or block. • Memory is allocated when the function or block is active and freed when the execution of the function or block is completed.	• Declared outside all the functions. • Available to all functions in the program. • Memory is allocated just before the execution of the program and freed when the program stops execution.
Functions	• Declared within a function or a block of statements and defined after the calling function. • Accessible only within that function or the block.	• Declared or defined outside all other functions. • Accessible by all functions in the program

Table 3.5: Scope and life of variables and functions

Let us conclude

Modular programming is an approach to make programming simpler. C++ facilitates modularization with functions. Function is a named unit of program to perform a specific task. There are two types of functions in C++: predefined functions and user-defined functions. Predefined functions can be used in a program only if we include the header file concerned in the program. User-defined functions may need to be declared if the definition appears after the calling function. During function call, data may be transferred from calling function to the called function through arguments. Arguments may be classified as actual arguments and formal arguments. Either call by value method or call by reference method can be used to invoke functions.

Lab activity

1. Define a function to accept a number and return 1 if it is odd, 0 otherwise. Using this function write a program to display all odd numbers between 10 and 50.
2. Write a program to find the product of three integer numbers using a user defined function. Invoke the function using call by value and call by reference methods. Verify the results.
3. Write a program to find the smallest of three or two given numbers using a function (use the concept of default arguments).
4. With the help of a user-defined function find the sum of digits of a number. That is if the given number is 345 then the result should be 3+4+5 = 12.
5. Using a function, write a program to find the area of a circle.
6. Write a program to check whether a number is positive, negative or zero. Use a user defined function for checking.

Let us assess

1. What is a meant by a function in C++?
2. The built-in function to find the length of a string is _____.
3. Write down the role of header files in C++ programs.
4. When will you use void data type for function definition?
5. Distinguish between actual parameters and formal parameters.
6. Construct the function prototypes for the following functions
   a. `Total()` takes two double arguments and returns a double
   b. `Math()` takes no arguments and has no return value
7. Discuss the scope of global and local variables with examples.
8. Distinguish between Call-by-value method and Call-by-reference method used for function calls.
9. In C++, function can be invoked without specifying all its arguments. How?
10. How a local function differs from a global function?
11. Identify the built-in functions needed for the following cases
    a. To convert the letter 'c' to 'C'
    b. To check whether a given character is alphabet or not.
    c. To combine strings "comp" and "ter" to make "computer"
    d. To find the square root of 25
    e. To return the number 10 from -10

12. Look at the following functions:

```
int sum(int a,int b=0,int c=0)
{
    return (a + b + c);
}
```

a. What is the speciality of the function regarding the parameter list?

b. Give the outputs of the following function calls by explaining its working and give reason if the function call is wrong.

i. `cout << sum (1, 2, 3);` ii. `cout << sum(5, 2);`

iii. `cout << sum();` iv. `cout << sum(0);`

13. The prototype of a function is: `int fun(int, int);`

The following function calls are invalid. Give reason for each.

a. `fun("hello",4);` b. `cout<<fun();` c. `val = fun(2.5, 3.3);`

d. `cin>>fun(a, b);` e. `z=fun(3);`

14. Consider the following program and predict the output if the radius is 5. Also write the reason for the output.

```
#include<iostream>
using namespace std;
float area(float &);
int main()
{
    float r, ans;
    cout<<"Enter radius :";
    cin>>r;
    ans = area(r);
    cout<<area;
    cout<<r;
    return 0;
}
float area(float &p)
{
    float q;
    q = 3.14 * p * p;
    p++;
    return q;
}
```

15. Modify the program given in question 14 by applying call by value method to call the function and write the possible difference in the output.