10

Functions

Key Concepts

- **Concept of modular programming**
- **Functions in C++**
- **Predefined functions**
  - String functions
  - Mathematical functions
  - Character functions
  - Conversion functions
  - I/O manipulating functions
- **User-defined functions**
  - Creating user-defined functions
  - Prototype of functions
  - Arguments of functions
  - Functions with default arguments
  - Methods of calling functions
- **Scope and life of variables and functions**
- **Recursive functions**
- **Creation of header Files**

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. As discussed in Chapter 4, 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 have already seen functions like `main()`, `sqrt()`, `gets()`, `getchar()`, etc. The functions, except `main()`, are assigned specific tasks and readily available for use. 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 more 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.

10.1 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 10.1.

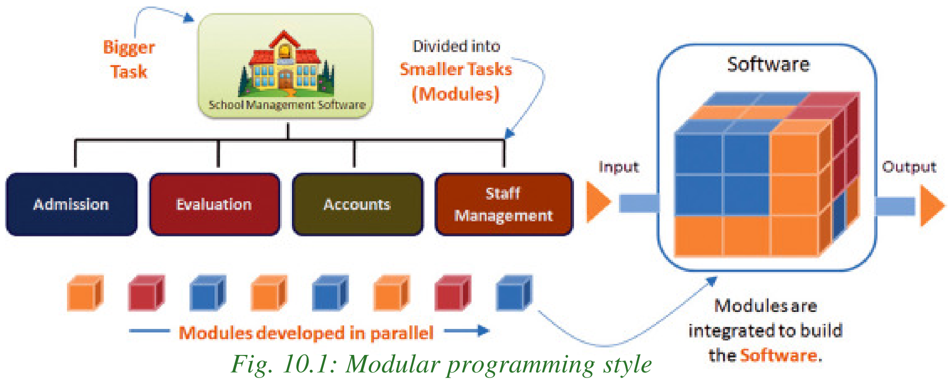

Bigger Task
School Management Software
Divided into Smaller Tasks (Modules)
Admission
Evaluation
Accounts
Staff Management
Modules developed in parallel
Input
Software
Output
Modules are integrated to build the Software.

Fig. 10.1: 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 a module and we write programs for each module. The process of breaking large programs into smaller sub programs is called **modularisation**. Computer programming languages have different methods to implement modularization. The sub programs are generally called functions. C++ also facilitates modular programming with functions.

Merits of modular programming

The modular style of programming has several 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 the 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 results 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 step 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 chances of error: When the size of the program is reduced, naturally syntax errors will be less in number. The chances of logical error will also be minimized. While solving complex problems we have to consider all the aspects of the problem, and hence the logic of the solution will also be complex. But in a modularized program, we need to concentrate only on one module at a time. If any error is identified in the output, we can identify the module concerned and rectify the error in that module only.

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. Thus modularization reduces the programming complexity by bringing down our mind to a simplified task at a time, reducing the program size and making the debugging process easy.

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.

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 the others. Utmost care should be taken while setting the hierarchy of the execution of the modules.

10.2 Functions in C++

Let us consider the case of a coffee making machine and discuss its functioning based on Figure 10.2. Water, milk, sugar and coffee powder are supplied to the machine. The machine processes it according to a set of predefined instructions stored in it and returns the coffee which is collected in a cup. The instruction-set may be as follows:

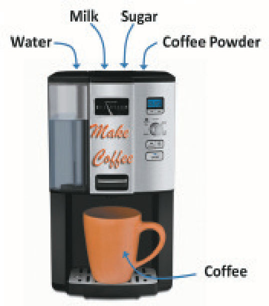


Fig. 10.2 : Functioning of a coffee making machine

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 10.3 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)

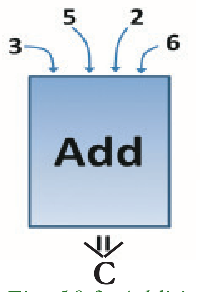


Fig. 10.3. Addition function

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 use for various tasks. The functions `clrscr()`, `getch()`, `sqrt()`, etc. are some of them. 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.

10.3 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 **value returned** by 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.

10.3.1 String Functions

Several string functions are available in C++ for the manipulation of strings. As discussed in Chapter 9, C++ does not have a string data type and hence an array of characters is used to handle strings. So, in the following discussion, wherever the word string comes, assume that it is a character array. Following are the commonly used string functions. We should include the header file **`cstring`** (`string.h` in Turbo C++) in our C++ program to use these functions.

a. `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 it will be stored in `n`.

b. `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` and `string2` are array of characters or 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:

```
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";
```

c. 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.

d. 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.

e. 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 the 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 the same during the comparison.

Program 10.1 compares and concatenates two strings. The length of the newly formed string is also displayed.

Program 10.1: To combine two strings if they are different and find its length

```
#include  <iostream>
#include  <cstring>
using  namespace  std;
int  main()
{
char  s1[20],  s2[20],  s3[20];
cout<<"Enter  two  strings:  ";
```

Header file essential for using string manipulating functions

```
cin>>s1>>s2;
int 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<<"\nString after concatenation is: "<<s3;
    cout<<"\nLength of the new string is: "<<strlen(s3);
}
return 0;
}
```

10.3.2 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.

a. 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.

b. 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;
```

c. 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, int)
```

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.

d. sin()

It is a trigonometric function and it finds the **sine** value of an angle. The argument to this function is `double` type data and it returns the sine value of the argument. The angle must be given in radian measure. Its syntax is:

```
double sin(double)
```

The sine value of $\angle 30°$ (*angle 30 degree*) can be found out by the following code.

```
float x = 30*3.14/180; //To convert angle into radians
cout << sin(x);
```

The above program code will display 0.4999770

e. cos()

The function is used to find the cosine value of an angle. The argument to this function is `double` type data and it returns the cosine value of the argument. In this case also, the angle must be given in radian measure. The syntax is:

```
double cos(double)
```

The cosine value of $\angle 30°$ (*angle 30 degree*) can be found out by the following code.

```
double x = cos(30*3.14/180);
cout << x;
```

Note that the argument provided is an expression that converts angle in degree into radians. The above code will display 0.866158.

Program 10.2 uses different mathematical functions to find the length of the sides of a right angled triangle, if an angle and length of one side is given. We use the following formula for solving this problem.

$$\text{Sin } \theta = \frac{\text{Opposite side}}{\text{Hypotenuse}}, \quad \text{Cos } \theta = \frac{\text{Adjacent side}}{\text{Hypotenuse}}, \quad \text{Tan } \theta = \frac{\text{Opposite side}}{\text{Adjacent side}}$$

$$(\text{Hypotenuse})^2 = (\text{Base})^2 + (\text{Altitude})^2$$

Program 10.2: To find the length of the sides of a right angled triangle using mathematical functions

```
#include <iostream>
#include <cmath>
```

Header file essential for using mathematical functions

```
using namespace std;
int main()
{
const float pi=22.0/7;
int angle, side;
float radians, length, opp_side, adj_side, hyp;
cout<<"Enter the angle in degree: ";
cin>>angle;
radians=angle*pi/180;
```

Conversion of angle in degree into radians

```
cout <<"\n1. Opposite Side"
     <<"\n2. Adjacent Side"
     <<"\n3. Hypotenuse";
cout <<"\nInput 1, 2 or 3 to specify the side: ";
cin>>side;
cout<<"Enter the length: ";
cin>>length;
switch(side)
{
    case 1: opp_side=length;
            adj_side=opp_side / tan(radians);
            hyp= sqrt(pow(opp_side,2) + pow(adj_side,2));
            break;
    case 2: adj_side=length;
            hyp=adj_side / cos(radians);
            opp_side=sqrt(pow(hyp,2) - pow(adj_side,2));
            break;
    case 3: hyp=length;
            opp_side=hyp * sin(radians);
            adj_side=sqrt(pow(hyp,2) - pow(opp_side,2));
}
cout<<"Angle in degree = "<<angle;
cout<<"\nOpposite Side = "<<opp_side;
cout<<"\nAdjacent Side = "<<adj_side;
cout<<"\nHypotenuse    = "<<hyp;
return 0;
}
```

The following is a sample output of the above program:

```
Enter the angle in degree: 30
1. Opposite Side
2. Adjacent Side
3. Hypotenuse
Input 1, 2 or 3 to specify the side: 1
Enter the length: 6
Angle in degree = 30
Opposite Side    = 6
Adjacent Side    = 10.38725
Hypotenuse       = 11.995623
```

10.3.3 Character functions

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

a. `isupper()`

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

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

The function returns 1 if the given character is in the 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.

b. `islower()`

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

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

The function returns 1 if the given character is in the 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 the 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');
```

c. 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');
```

d. 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);
```

e. isalnum()

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

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

The function returns 1 if the given character is an 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 an alphabet nor a digit.

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

f. `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 the 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'`.

g. `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 the 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 are executed, the value of the variable `c` will be `'a'`.

```
char x = ‘a’;
char c = tolower(x) ;
```

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

Program 10.3 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 the upper and lower cases.

Program 10.3: 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, i;
cout << "Enter a line of text: ";
gets(text);
for(i=0; text[i]!='\0'; i++)
    if (isupper(text[i])) Ucase++;
         else if (islower(text[i])) Lcase++;
              else if (isdigit(text[i])) Digit++;
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;
}
```

Loop will be terminated when the value of `i` points to the null character is reached

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
```

The input by the user

10.3.4 Conversion functions

These functions are used to convert a string to integer and an integer to string. Following are the different conversion functions available in C++. The header file **`cstdlib`** (**`stdlib.h`** in Turbo C++) is to be included to use these functions in a program.

a. `itoa()`

This function is used to convert an integer value to string type. The syntax of the function is:

```
    itoa(int n, char c[], int len)
```

From the syntax, we can see that the function requires three arguments. The first one is the number to be converted. The second argument is the character array where the converted string value is to be stored and the last argument is the size of the character array. The following code segment illustrates this function:

```
    int n = 2345;
    char c[10];
    itoa(n, c, 10);
    cout << c;
```

The above program code will display `"2345"` on the screen.

b. `atoi()`

This function is used to convert a string value to integer. The syntax of the function is:

```
    int atoi(char c[]);
```

The function takes a string as argument returns the integer value of the string. The following code converts the string `"2345"` into integer number 2345.

```
    int n;
    char c[10] = "2345";
    n = atoi(c);
    cout << n;
```

If the string consists of characters other than digits, the output will be 0. But if the string begins with digits, only that part will be converted into integer. Some usages of this function and their outputs are given below:

(i) `atoi("Computer")` returns 0

(ii) `atoi("12.56")` returns 12

(iii) `atoi("a2b")` returns 0

(iv) `atoi("2ab")` returns 2

(v) `atoi(".25")` returns 0

(vi) `atoi("5+3")` returns 5

Program 10.4 illustrates the use of these functions in problem solving. It accepts the three parts (day, month and year) of a date of birth and displays it in date format.

Program 10.4: To display date of birth in date format

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
int main()
{
char dd[10], mm[10], yy[10], dob[30];
int d, m, y;
cout<<"Enter day, month and year in your Date of Birth: ";
cin>>d>>m>>y;
itoa(d, dd, 10);
itoa(m, mm, 10);
itoa(y, yy, 10);
strcpy(dob, dd);
strcat(dob, "-");
strcat(dob, mm);
strcat(dob, "-");
strcat(dob, yy);
cout<<"Your Date of Birth is "<<dob;
return 0;
}
```

A sample output of Program10.4 is given below:

```
Enter day, month and year in your Date of Birth: 26  11  1999
Your Date of Birth is 26-11-1999
```

10.3.5 I/O Manipulating function

These functions are used to manipulate the input and output operations in C++. The header file **ciomanip** is to be included to use these functions in a program.

setw()

This function is used to set the width for the subsequent string. The following code shows its impact in the output:

```
char  s[]="hello";
cout<<setw(10)<<s<<setw(10)<<"friends";
```

The output of the above code will be as follows:

```
hello    friends
```

The word `hello` will be displayed right aligned within a span of 10 character positions. Similarly, the word `friends` will also be displayed right aligned within a span of 10 character positions.

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

Check yourself

1. What is modular programming?
2. What is a function in C++?
3. Name the header file required for using character functions.
4. Name the function that displays the subsequent data in the specified width.
5. Pick the odd one out and give reason:
   (a) `strlen()` (b) `itoa()` (c) `strcpy()` (d) `strcat()`

10.4 User-defined functions

All the programs that we have discussed so far contain a function named **main()**. We know that the first line is a pre-processor directive statement. The remaining part is actually the definition of a function. The `void 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 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 `getchar()`, `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()` does not take any argument. 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. But some functions like `setw()` 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.

10.4.1 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

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

Function 2

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

Function 3

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

Function 4

```
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 10.1 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 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. In most of the functions,

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 10.1 : Analysis of functions

`return` is placed at the end of the function. The functions defined with void data type may have a `return` statement within the body, but we cannot provide any value to it. The return type of `main()` function is either `void` or `int`.

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"
}
```

The output of the program is 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 check whether a given number is perfect or not. A number is said to be perfect if it is equal to the sum of its factors other than the number itself. For example, 28 is a perfect number, because the factors other than 28 are 1, 2, 4, 7 and 14. Sum of these factors is 28. For solving this problem, let us define a function that accepts a number as argument and returns the sum of its factors. Using this function, we will write the program. 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 10.5 - Perfect number checking - Program 10.6

Function before main()

```
#include <iostream>
using namespace std;
int sumfact(int N)
{ int i, s = 0;
  for(i=1; i<=N/2; i++)
     if (N%i == 0)
        s = s + i;
  return s;
}
//Definition above main()
int main()
{
  int num;
  cout<<"Enter the Number: ";
  cin>>num;
  if (num==sumfact(num))
     cout<<"Perfect number";
  else
     cout<<"Not Perfect";
  return 0;
}
```

Function after main()

```
#include <iostream>
using namespace std;
int main()
{
  int num;
  cout<<"Enter the Number: ";
  cin>>num;
  if (num==sumfact(num))
     cout<<"Perfect number";
  else
     cout<<"Not Perfect";
  return 0;
}
//Definition below main()
int sumfact(int N)
{ int i, s = 0;
  for(i=1; i<=N/2; i++)
     if (N%i == 0)
        s = s + i;
  return s;
}
```

When we compile Program 10.5, there will be no error and we will get the following output on execution:

```
Enter the Number: 28
Perfect number
```

If we compile Program 10.6, there will be an error `'sumfact was not declared in this scope'`. Let us see what this error means.

10.4.2 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 10.5 contains the `main()` function after the user-defined function, but in Program 10.6, the `main()` is defined before the user-defined function. When we compile this program, it will give an error - `"sumfact was not declared in this scope"`. This is because the function `sumfact()` is called in the program, before it is defined. During the compilation of the `main()` function, when the compiler encounters the function call `sumfact()`, it is not aware of such a function. Compiler is unable to check whether there is such a function and whether its usage is correct or not. So it reports an error arising out of the absence of the function 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.

So, the error in Program 10.6 can be rectified by inserting the following statement before the function call in the `main()` function.

```
    int  sumfact(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()`.

10.4.3 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(long 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 10.7: To find the value of nCr

```
#include<iostream>
using namespace std;
int fact(int);
int main()
{
    int n,r;
    int 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)
{
    int f;
    for(f=1; N>0; N--)
        f=f*N;
    return f;
}
```

Function prototype

Function call according to the formula

Actual arguments

Function header

Formal argument

Factorial is returned

The following is a sample output:

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

User inputs

10.4.4 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.

```
    long TimeSec(int H, int M=0, int S=0)
    {
        long 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.

10.4.5 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 list of names in two ways. She can take a photocopy of the list and give it to you. Otherwise, she can give the original 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 mark or write anything in the name list because the teacher may want the same name list for future use. But if you are given a photocopy of the list, you can mark or write 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';
}
int 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 10.2 shows what happens to the arguments when a function is called using call-by-value method:

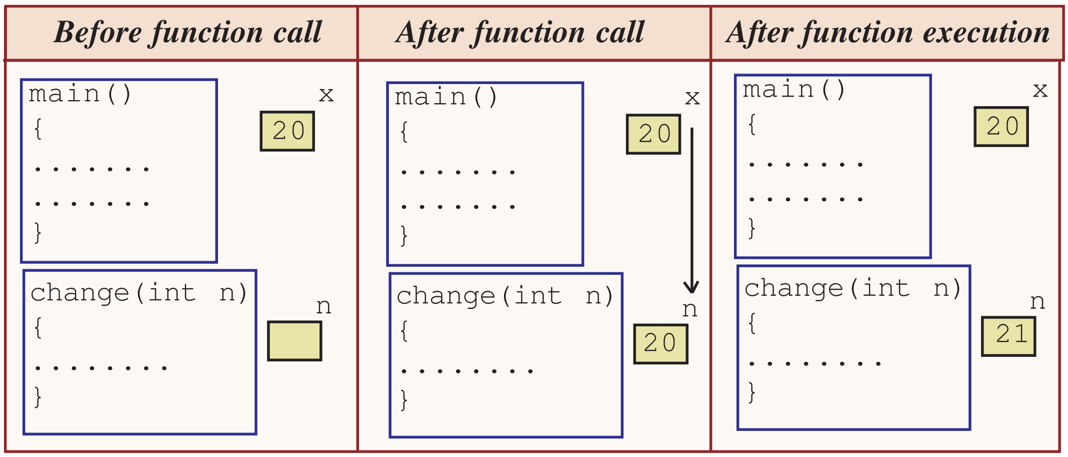


Table 10.2: Call by value procedure

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

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

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

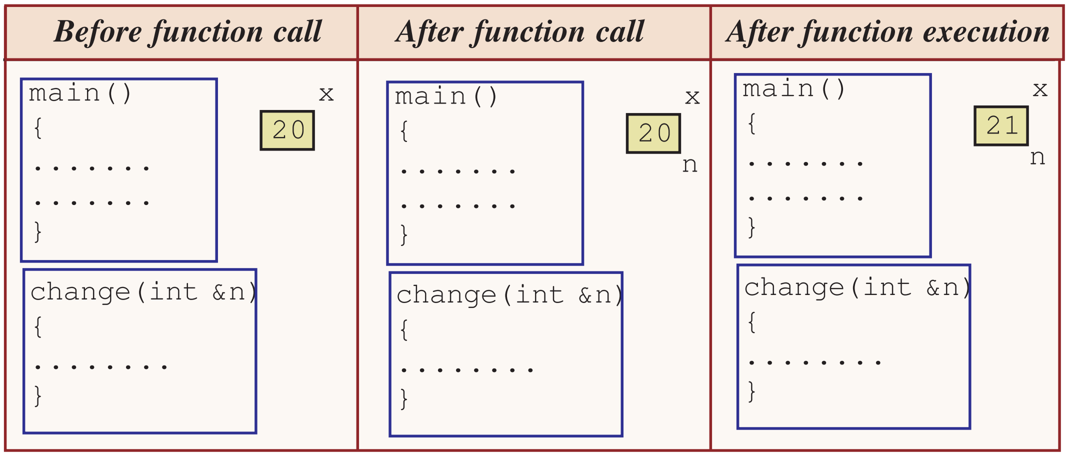


Table 10.3: Call by reference procedure

These two methods of function call differ as shown in Table 10.4.

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

Table 10.4 : 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 10.8: 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 10.8. 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.

Check yourself

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++?

10.5 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 10.9 illustrates the accessibility of local variables in a program.

Program 10.9: 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;
}
int main()
{
    int x, result;
    cout << "Enter a number : ";
    cin >> x;
    result = cube(x);
    cout << "Cube = " << result;
    cout << "\nCube = "<<cb;
}
```

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

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 10.10 to get more clarity on the scope and life of variables and functions throughout the program.

Program 10.10 : 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;
}
```

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 the functions of 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 the functions in the program

Table 10.5 : Scope and life of variables and functions

The given call-outs 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 10.5 summarises the scope and life time of variables and functions.

10.6 Recursive functions

Usually a function is called by another function. Now let us define a function that calls itself. The process of calling a function by itself is known as **recursion** and the function is known as **recursive function**. Some of the complex algorithms can be easily simplified by using the method of recursion. A typical recursive function will be as follows:

```
int function1()
{
    ............
    ............
    int n = function1();    //calling itself
    ............
    ............
}
```

In recursive functions, the function is usually called based on some condition only. Following is an example for recursive function by which the factorial of a number can be found. Note that factorial of a negative number is not defined. Therefore 'n' can take the value either zero or a positive integer.

```
int factorial(int n)
{
    if((n==1)||(n==0))
        return 1;
    else if (n>1)
        return (n * factorial(n-1));
    else
        return 0;
}
```

Let us discuss how this function is executed when the user calls this function as:

```
f = factorial(3);
```

When the function is called for the first time, the value of the parameter `n` is 3. The condition in the `if` statement is evaluated to be false. So, the `else` block is executed.

When the value of `n` is 3, the instruction in the `else` block becomes

```
        return  (3  *  factorial(3-1));
```

which is simplified as: `return  (3 * factorial(2));` .... (i)

In order to find `3 * factorial(2)`, it needs to find the value of `factorial(2)`. The `factorial()` function is called again with 2 as the argument. The function is called within the same function. When the `factorial()` function is executed with 2 as the value of the parameter `n`, the condition in the `if` block becomes false. So only the `else` block is executed. The instruction in the `else` block is:

```
        return  (2  *  factorial(2-1));
```

which is simplified as: `return  (2 * factorial(1));` .... (ii)

In order to find this returning value, it calls the `factorial()` function again with 1 as the value of the parameter `n`. Now, the condition in the `if` statement becomes true, hence it will return 1 as the value of the function call `factorial(1);`

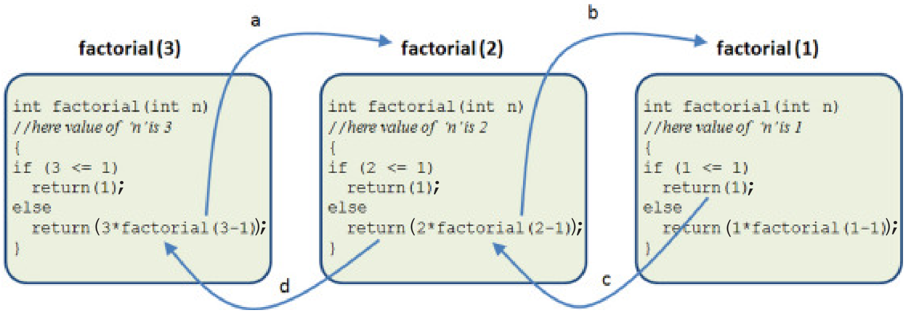
Now, the program can find the return value in the instruction numbered (ii). The instruction (ii) will become: `return 2 * 1;`
which is same as: `return 2;`

That is, the value 2 is returned as the value of the function call `factorial(2)` in the instruction numbered (i). Thus the instruction (i) becomes: `return 3 * 2;`
which is same as: `return 6;`

Now the value 6 is returned as the value of the function call `factorial(3);`.

The execution of the instruction `f=factorial(3);` is summarised in Figure 10.4.

The execution of the function call `factorial(3)` is delayed till it gets the value of the function `factorial(2)`. The execution of this function is delayed till it gets the value of the function `factorial(1)`. Once this function call gets 1 as its return value, it returns the value to the previous function calls. Figure 10.4 shows what happens to the arguments and return value on each call of the function.



```
int factorial(int n)
//here value of 'n'is 3
{
if (3 <= 1)
  return(1);
else
  return(3*factorial(3-1));
}
```

```
int factorial(int n)
//here value of 'n'is 2
{
if (2 <= 1)
  return(1);
else
  return(2*factorial(2-1));
}
```

```
int factorial(int n)
//here value of 'n'is 1
{
if (1 <= 1)
  return(1);
else
  return(1*factorial(1-1));
}
```

Fig. 10.4: Control flow in a recursive function call

Note that if we call the function `factorial()` with a negative integer as argument, the function will return 0. It doesn't mean that the factorial of a negative number is 0. In mathematics the factorial of a negative number is undefined. So, for example, the value returned by the function call `factorial(-3)` should be interpreted as an invalid call in the calling function.

The procedure that takes place when the function `factorial()` is called with 5 is shown in Figure 10.5.

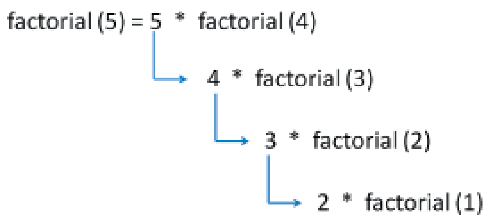


Fig. 10.5: Recursion process for factorial(5)

A function to find the factorial of a number can also be defined as follows, without using recursion.

```
int factorial(int n)
{
    int f=1;
    /*  The formula n×(n-1)×(n-2)× ... ×2×1 is appliled
    instead of 1 × 2 × 3× ... ×(n-1)×n to find the factorial
    */
    for(int i=n; i>1; i--)
        f *= i;
    return f;
}
```

We can compare the difference between the two functions – the one that uses recursion and the other that does not use recursion. Every function that uses recursion can also be written without using recursion. Then why do we use recursion? Some programmers find the use of recursion much simpler than the other. The recursion method cannot be used in all the functions. How do we understand whether recursion can be applied to a function or not? If we can express a function by performing some operation on the output of the same function, we can use recursion for that. For example, to find the sum of first n natural numbers, we can write the `sum(n)` as:

```
    sum(n) = n + sum(n-1)
```

Let us discuss a program that converts a given decimal number into its equivalent binary number. We discussed the conversion procedure in Chapter 2.

Program 10.11: To display the binary equivalent of a decimal number

```
#include <iostream>
using namespace std;
void Binary(int);
int main()
{
    int decimal;
    cout<<"Enter an integer number: ";
    cin>>decimal;
    cout<<"Binary equivalent of "<<decimal<<" is ";
    Binary(decimal);
    return 0;
}

void Binary(int n)    //Definition of a recursive function
{
    if (n>1)
        Binary(n/2);
    cout<<n%2;
}
```

A sample output of Program 10.11 is given below:

```
    Enter an integer number: 19
    Binary equivalent of 19 is 10011
```

10.7 Creation of header files

All of us know that we always use `#include <iostream>` in the beginning of all the programs that we have discussed so far. Why do we use this statement? Actually `iostream` is a header file that contains the declarations and definitions of so many variables or objects that we use in C++ programs. The objects `cout` and `cin` that we use in the program are declared in this header file. So when we include this header file in a program, the definitions and declarations of objects and functions are available to the compiler during compilation. Then the executable code of these functions and objects will be linked with the program and will be executed when and where they are called. We can create similar header files containing our own variables and functions. Suppose we want to write a function to find the factorial of a number and use this function in a number of programs. Instead of defining the factorial function in all the programs, we can place the function in a header file and then include this header file in all other programs.

The following example shows how we can create a header file. Enter the following program code in any IDE editor.

```
    int factorial(int n)
    {
        int f=1;
        for (int i=1; i<=n; i++)
            f *= i;
        return f;
    }
```

Save this in a file with the name **`factorial.h`** and then create a C++ program as follows:

```
#include <iostream>
#include "factorial.h"//includes user-defined headerfile
using namespace std;
int main()
{
    int n;
    cout<<"Enter a number : ";
    cin >> n;
    cout<<"Factorial : " << factorial(n);
}
```

We can compile and run the program successfully. Note that the statement `#include "factorial.h"` uses double quotes instead of angular brackets < >. This is because, when we use angular brackets for including a file, the compiler will search for the file in the include directory. But if we use double quotes, the compiler will search for the file in the current working directory only. Usually when we save the `factorial.h` file, it will be saved in the working directory, where the main C++ program is saved. So, we must use double quotes to include the file. Now, whenever we want to include factorial function in any C++ program, we only need to include header file `factorial.h` in the program by using the statement `#include "factorial.h"`. In the same way we can place any number of functions in a single header file and include that header file in any program to make use of these functions.

Check yourself

1. If the prototype of a function is given immediately after the preprocessor directive, its scope will be ________.
2. What is recursion?
3. What is the scope of predefined functions in C++?
4. The arguments of a function have _______ scope.

Let us sum up

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. The variables and functions in a program have scope and life depending on the place of their declaration. Though a function is called by another function, C++ allows recursion which is a process of calling a function by itself. New header files can be created to store the user-defined functions so that these functions can be used in any program.

Learning outcomes

After completing this chapter the learner will be able to

- recognise modular programming style and its merits.
- use predefined functions for problem solving.
- define sub functions for performing particular tasks involved in problem solving.
- use the sub functions defined by the user.
- define the recursive functions and use them for problem solving.

Lab activity

1. Define a function to accept a number and return 1 if it is prime, 0 otherwise. Using this function write a program to display all prime numbers between 100 and 200.
2. Write a program to find the smallest of three or two given numbers using a function (use the concept of default arguments).
3. With the help of a user-defined function find the sum of digits of a number. That is, if the given number is 3245, the result should be 3+2+ 4+5 = 14.
4. Using a function, write a program to find the LCM of two given numbers.
5. Write a program to display all palindrome numbers between a given range using a function. The function should accept number and return 1 if it is palindrome, 0 otherwise.

Sample questions

Very short answer type

1. How do top down design and bottom up design differ in programming?
2. What is a function in C++?
3. The ability of a function to call itself is __________.
4. Write down the role of header files in C++ programs.
5. When will you use void data type for function definition?

Short answer type

1. Distinguish between actual parameters and formal parameters.
2. 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
3. Distinguish between `exit()` function and `return` statement.
4. Discuss the scope of global and local variables with examples.
5. Distinguish between Call-by-value method and Call-by-reference method used for function calls.
6. In C++, function can be invoked without specifying all its arguments. How?
7. Write down the process involved in recursion.

Long answer type

1. 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);`
2. The prototype of a function is: `int fun(int, int);`
   The following function calls are invalid. Give reason for each.
   (a) `fun(2,4);` (b) `cout<<fun();` (c) `val=fun(2.5, 3.3);`
   (d) `cin>>fun(a, b);` (e) `z=fun(3);`