

Functions in C++

7

CHAPTER

CHAPTER OUTLINE

- 7.1 Introduction
- 7.2 Parts of a Function
- 7.3 Passing Arguments
- 7.4 Lvalues and Rvalues
- 7.5 Return by Reference
- 7.6 Returning More Values by Reference
- 7.7 Default Arguments
- 7.8 `const` Arguments
- 7.9 Inputting Default Arguments
- 7.10 Inline Functions
- 7.11 Function Overloading
- 7.12 Principles of Function Overloading
- 7.13 Precautions with Function Overloading
- 7.14 Recursion
- 7.15 Library Functions
- 7.16 More Programs

7.1 INTRODUCTION

In our daily life, most of the complex works are done by a group of people. This results in better productivity and quality. Work is done at a great speed. Normally, big work is distributed among many people. For example, to declare the result of examination in an autonomous institute, marks/score of all subjects, practical, seminar, projects, etc., are collected from different departments through professors and the result is compiled, declared, and displayed.

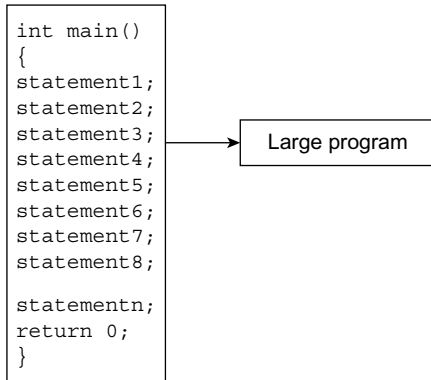


Fig. 7.1 Program without functions

The same concept is applied while developing complex programs/software applications using programming languages. In case the size of the program is large, it becomes difficult to maintain it and it is also hard to identify the flow of data. It is preferred to divide the large program into small modules called functions. Each function takes the data that are provided by the `main()` function and carries out operations as per the requirement, and results can be written to the calling function. Thus, programs are made more efficient.

When the concept of function or sub-program was not introduced, the program sizes were large and few codes get repeated as shown in the Figure 7.1.

It was very difficult to debug and update these large programs because many bugs get encountered. When the functions and sub-programs are introduced, the programs are divided into a number of segments and code duplication is avoided as shown in Figure 7.2. Bugs in small programs can be searched easily and this step leads to the development of complex programs with functions.

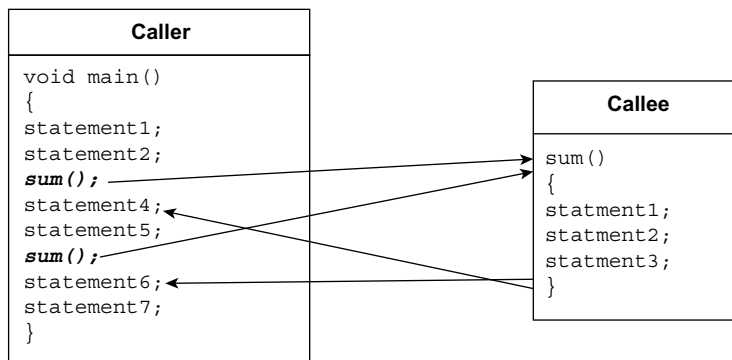


Fig. 7.2 Program with function

One of the features of C/C++ language is that a large program can be divided into smaller sub-programs. The smaller programs can be written in the form of functions. The process of dividing a large program into tiny and handy sub-programs and manipulating them independently is known as **modular programming**. This technique is similar to divide-and-conquer technique. Dividing a large program into smaller functions provides advantages to the programmer. Testing and debugging a program with functions is easier. A function can be called repeatedly based on the application and thus the size of the program can be reduced. The message passing between a caller (calling function) and callee (called function) takes place using arguments. The concept of modular approach of C++ is obtained from a function. Advantages of the functions are as follows:

- **Reusability:** A function once written can be invoked again and again, thus helping us to reuse the code and removing data redundancy.

- **Modularity:** Functions can help us in breaking a large, hard to manage problem into smaller manageable sub-problems. It is easier to understand the logic of sub-programs.
- **Reduced Program Size:** Functions can reduce the size of the program by removing data redundancy.
- **Easy Debugging:** Using functions, debugging of a program becomes very easy, as it is easier to locate and rectify the bug in the program if functions are used.
- **Easy Updating:** If we need to update some code in the program, then it is much more easier in case we have used functions, as the changes need to be made in one place only (in function).

The programs written in 'C++' language are highly depending on functions. The 'C++' program is nothing but a combination of one or more functions. Execution of every 'C++' program starts with user defined function `main()`. The method of using functions is slightly changed and enhanced in C++ as compared to C. The C++ language adds few new features to function like overloading of functions, default arguments etc. These features are described in ahead.

C++ functions are classified in two categories. They are (1) library functions and (2) user-defined functions. The library functions can be used in any program by including respective header files. The header files must be included using `#include` pre-processor directive. For example, a mathematical function uses `math.h` header file.

The programmer can also define and use his/her own functions for performing some specific tasks. Such functions are called *user-defined functions*. Every C++ program consists of `main()` function. This is a user-defined function in which program statements are written by the programmer according to the problem definition.

7.2 PARTS OF A FUNCTION

Parts of a function are as follows.

- (a) Function prototype declaration
- (b) Function call
- (c) Definition of a function
- (d) Actual and formal arguments
- (e) Return statement

7.2.1 Function Prototype Declaration

We use many built-in functions. The prototypes of these functions are given in the respective header files, and we include them using `#include` directive. In C++, while defining user-defined functions, it is unavoidable to declare its prototype. A prototype statement helps the compiler to check the return and argument types of the function.

A function prototype declaration consists of function's return type, name, and arguments list. It tells the compiler

- (a) Name of the function
- (b) Type of value returned
- (c) The type and number of arguments

When the programmer defines the function, the definition of function must be same like its prototype declaration. If programmer makes mistake, the compiler flags an error message. The function

prototype declaration statement is always terminated with semicolon. The statements given below are the examples of function prototypes.

```
(A) void show (void);
(B) float sum (float, int);
(C) float sum (float x, int y);
```

In example (A) the return type is void, that is the function does not return any value. The void functions are always without return statement. The void argument, that is the function, does not require any argument. By default, every function returns an integer value. To return a non-integer value, the data type should be mentioned in function prototype and definition. While writing definition of function, the return type must be preceded by the function name and it is optional if return type is default (int).

In statement (B), the prototype of function sum() is declared. Its return type is float and arguments are float and integer type, respectively. It is shown in Figure 7.3.

In example (C) with argument type, argument names are also declared. It is optional and also not compulsory to use the same variable name in the program.

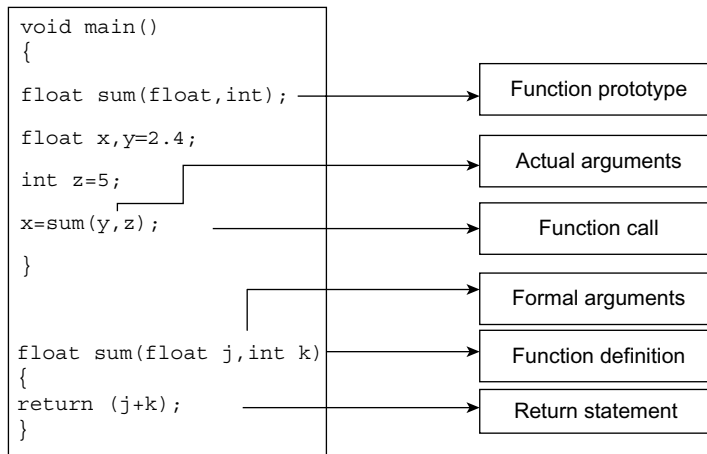


Fig. 7.3 Parts of a function

7.2.2 Function Call

A function is a latent body. It gets activated only when a call to a function is invoked. A function must be called by its name followed by argument, or without argument, list enclosed in parenthesis and terminated by semicolon.

Syntax of function call is as follows:

```
function-name (with/without argument list);
```

In the above statement, function-name is the name of the function, arguments are within the bracket and arguments are separated by comma. If arguments are absent one can write void within the bracket.

7.2.3 Function Definition

The first line is called **function definition** and function body follows it. The function definition and function prototype should match with each other. The function body is enclosed within curly braces. The function can be defined anywhere. If the function is defined before its caller, then its prototype declaration is optional.

Syntax of function call is as follows:

```
return_data_type function-name(argument/parameter list);
{
variable declarations
function statements
}
```

Return data type specifies the type of value returned by the function.

The function-name is the name of the function being defined.

The argument/parameter list specifies types and names of the arguments/parameters (also called formal arguments).

7.2.4 Actual and Formal Argument

The arguments declared in caller function and given in the function call are called **actual arguments**. The arguments declared in the function definition are known as **formal arguments**.

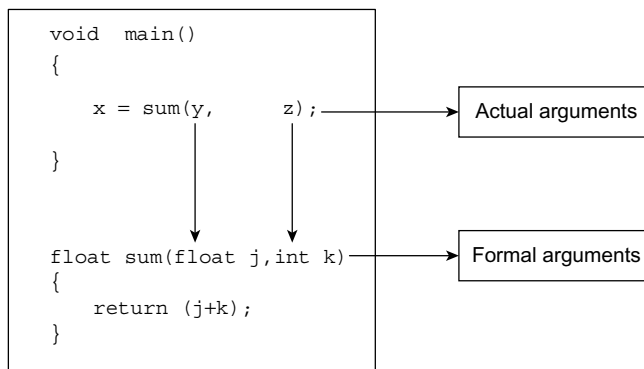


Fig. 7.4 Actual and formal arguments

As shown in Figure 7.4, variables `y` and `z` are actual arguments and variables `j` and `k` are formal arguments. The values of `y` and `z` are stored in `j` and `k`, respectively. The values of actual arguments are assigned to formal arguments. The function uses formal arguments for computing.

7.2.5 The return Statement

The return statement is used to return value to the caller function. The return statement returns only one value at a time. When a return statement is encountered, compiler transfers the control of the program to caller function. The syntax of return statement is as follows:

```
return (variable name); or return variable name;
```

The parenthesis is optional.

Few programs are as follows:

7.1 Write a program to declare prototype of function `sum()`. Define the function `sum()` exactly similar to its prototype.

```
#include<iostream.h>
#include<conio.h>

int main()
{
    clrscr();
    float sum (int, float ); // function prototype
    int a=20;
    float s,b=2.5;
    s=sum(a,b);
    cout<<"Sum=" <<s;
    return 0;
}

float sum (int x, float y)
{
    return (x+y);
}
```

OUTPUT

Sum = 22.5

Explanation: In the above program, the prototype of function `sum()` is declared. The prototype instructs the compiler that the function `sum()` should return `float` value. The types of arguments used by function `sum()` are `int` and `float`, respectively. The values of variables `a` and `b` are passed to function `sum()`. These values are assigned to formal arguments `x` and `y`. The sum of `x` and `y` is calculated and returned. The return value of function is assigned to `float` variable `s`.

7.2 Write a program to declare and define void function.

```
#include<iostream.h>
#include<conio.h>

int main()
{
    clrscr();
    void show (void);
    show();
    return 0;
}

void show()
{
    // ...
}
```

```
    cout<<"\n In show() ";  
}
```

OUTPUT

In show()

Explanation: In the above program, the prototype of function `show()` is given preceding `void` keyword. The statement `show()` invokes the `show()` function that displays the message “In show()”.

7.3 PASSING ARGUMENTS

The main objective of passing argument to function is message passing. The message passing is also known as communication between two functions, that is between caller and called functions. There are three methods by which we can pass values to the function. These methods are as follows:

- (a) Call by value (pass by value)
- (b) Call by address (pass by address)
- (c) Call by reference (pass by reference)

C++ supports all these three types of passing values to the function, whereas C supports the first two types only. The arguments used to send values to function are known as input arguments. The arguments used to return result are known as output arguments. The arguments used to send as well as return results are known as input-output arguments. While passing values to the function, the following conditions should be fulfilled.

The data type and number of actual and formal arguments should be same both in caller and callee functions. Extra arguments are discarded if they are declared. If the formal arguments are more than the actual arguments, then the extra arguments appear as garbage. Any mismatch in the data type will produce the unexpected result.

7.3.1 Call by Value

In this type, values of actual arguments are passed to the formal arguments and operation is done on the formal arguments. Any change in the formal arguments does not effect to the actual arguments because formal arguments are photocopy of actual arguments. Hence, when function is called by call by value method, it does not affect the actual contents of actual arguments. Changes made in the formal arguments are local to the block of called function. Once control returns back to the calling function, the changes made will vanish. The advantage of this call by value is that actual parameters are fully protected because their values are not changed when control is returned to the calling function. The following example illustrates the use of call by value.

7.3 Write a program to demonstrate call by value.

```
#include<iostream.h>  
#include<conio.h>
```

```

int main()
{
    clrscr();
    int x,y;
    void swap (int, int);
    cout<<"\n Enter Values of X & Y:";
    cin>>x>>y;
    swap(x,y);
    cout<<"\n\n In function main() ";
    cout<<"\n Values X="<<x <<" and Y= "<<y;
    cout<<"\nAddress X="<<(unsigned) &x <<" and Y= "<<(unsigned) &y;
    return 0;
}
void swap(int a, int b)
{
    int k; k=a; a=b; b=k;
    cout<<"\n In function swap() ";
    cout<<"\n Values X="<<a <<" and Y= "<<b;
    cout<<"\nAddress X="<<(unsigned) &a <<" and Y=
"<<(unsigned) &b;
}

```

OUTPUT

Enter Values of X & Y :5 4

In function swap()

Values X=4 and Y= 5

Address X=4090 and Y= 4092

In function main()

Values X=5 and Y= 4

Address X=4096 and Y= 4094

Explanation: In the above program we are passing values of actual arguments 'x' and 'y' to function swap(). The formal arguments 'a' and 'b' of function swap() receive these values. The values are exchanged i.e. value of 'a' is assigned to 'b' and vice-versa. They are displayed on the screen. When the control returns back to the main(), the changes made in function swap() vanish, because a and b are local variables of function swap(). In the main() the values of 'x' and 'y' are printed as they read from the keyboard. In call by value method the formal argument acts as duplicate of the actual argument. The addresses of actual and formal arguments are different. Thus, changes made with the variables are temporary.

7.3.2 Call by Address

In this type, instead of passing values, addresses of actual parameters are passed to the function by using pointers. Function operates on addresses rather than values. Here the formal arguments

are pointers to the actual arguments. In this type, formal arguments are pointers to actual argument. Because of this, when the values of formal arguments are changed, the values of actual parameters also change. Hence changes made in the argument are permanent. The following example illustrates passing the arguments to the function using call by address method.

7.4 Write a program to demonstrate pass by address.



```
#include<iostream.h>
#include<conio.h>

int main()
{
    clrscr();
    int x,y;
    void swap (int *, int *);
    cout<<"\n Enter Values of X & Y:";
    cin>>x>>y;
    swap(&x, &y);
    cout<<"\n In main()";
    cout<<"\n Values X="<<x <<" and Y="<<y;
    cout<<"\n Address X="<<(unsigned) &x <<" and Y="<<(unsigned) &y;
    return 0;
}

void swap(int *a, int *b)
{
    int *k;
    *k=*a;
    *a=*b;
    *b=*k;
    cout<<"\n In swap()";
    cout<<"\n Values X="<<*a <<" and Y="<<*b;
    cout<<"\n Address X="<<(unsigned) a <<" and Y="<<(unsigned) b;
}
```

OUTPUT

```
Enter Values of X & Y :5 4
In swap()
Values X=4 and Y=5
Address X=4096 and Y=4094
In main()
Values X=4 and Y=5
Address X=4096 and Y=4094
```

Explanation: In the above example we are passing addresses of actual arguments to the function swap (). The formal arguments are declared as pointers in the function declaration of swap ().

The formal arguments receive addresses of actual arguments i.e. formal arguments points to the actual argument. Here the `swap()` function operates on the addresses of actual argument through pointers. The addresses of actual arguments and formal arguments are same. Hence, the changes made in the values are permanent.

7.3.3 Call by Reference

C passes arguments by value and address. In C++ it is possible to pass arguments by value, address, and reference. C++ reference types, declared with `'&'` operator, are nearly identical but not exactly the same as pointer types. They declare aliases for objects variables and allow the programmer to pass arguments by reference to functions. The reference decelerator (`&`) can be used to declare references outside functions.

```
int k = 0;
int &kk = k; // kk is an alias for k
kk = 2; // same effect as k = 2
```

This creates the lvalue `kk` as an alias (assumed name) for `k`, provided that the initializer is the same type as the reference. Any operations on `kk` have exactly the same result as operations on `k`.

```
Example,
kk = 5 // assigns 5 to kk and
&kk // returns the address of kk.
```

The reference decelerator can also be used to declare reference type parameters within a function:

```
void funcA (int i);
void funcB (int &kk); // kk is type "reference to int"
int s=4;
funcA(s); // s passed by value
funcB(s); // s passed by reference
```

The `s` argument passed by reference can be modified directly by `funcB`; whereas the function `funcA` receives a duplicate copy (not actual) of the `s` argument (passed by value). Due to this reason, variable `s` itself cannot be modified by `funcA`. When an actual variable `s` is not appearing in the above example, argument `s` is passed by value, and the matching formal argument in the function obtains a copy of `s`. Any alterations to this copy inside the scope of the function body are not thrown back in the value of `s` itself. Absolutely, the function can return a value that could be used later to change `s`, but the function cannot directly alter a parameter passed by value. The conventional C method for changing `s` operates on the address of actual argument (`&s`), the address of `s`, rather than `s` itself. Although `&s` is passed by value, the function can access `s` through the copy of `&s` it receives. Even though the function does not need to alter `s`, it is still useful (though subject to possibly risky secondary results) to pass `&s`, especially if `s` is a large data structure. Passing `s` directly by value contains the useless copying of the data structure.

7.5 Write a program to pass the value of variable by value, reference, and address and display the result.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>

int main()
{
    clrscr();
    void funA (int s);
    void funB (int & );
    void funC (int *);
    int s=4; // initial value
    funA(s); // s passed by value
    cout<<"\nValue of s= "<<s <<" Address of s: "
        <<unsigned(&s);
    funB(s); // s passed by reference
    cout<<"\nValue of s= "<<s <<" Address of s: "
        <<unsigned(&s);
    funC(&s); // s passed by reference (C style )
    cout<<"\nValue of s= "<<s <<" Address of s: "
        <<unsigned(&s);
    return 0;
}

void funA ( int i)
{ i++;}
void funB (int &k)
{ k++;}
void funC(int *j)
{ ++*j;}
```

OUTPUT

Value of s= 4 Address of s : 4096

Value of s= 5 Address of s : 4096

Value of s= 6 Address of s : 4096

Explanation: In the above example, funA(), funB(), and funC() functions are declared. The integer variable s is declared and initialized to 4. The funA() is invoked and value of s is passed by value. In function funA() the value of s is incremented. After execution of funA() the value of s printed in main() is the same as the previous one. Thus, passing variable by value cannot change the contents of the variable.

The value of s is once again passed to funB(). The function funB() receives the value of s by reference. The value of s is received by the variable k by reference, that is variable k is

an alias for variable *s* and both have the same memory location. The variable *k* is incremented. Thus, any change made in reference variable affects the actual variable. Thus, after execution of `funB()` the printed value of *s* is 5.

The third function `funC()` uses conventional C style. Here, address of the variable *s* is passed to `funC()`. The formal argument `*j` is pointer to the actual argument. Thus, any change made through pointer *j* also reflects on the actual variable. Thus, we can change the value of variable using call by reference. Figure 7.5 explains the methods of passing arguments to function.

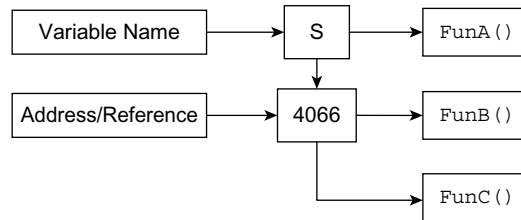


Fig. 7.5 Argument passing methods

7.6 Write a program to demonstrate call by reference.

```

#include<iostream.h>
#include<conio.h>

int main()
{
    clrscr();
    int x,y;
    void swap (int &, int &);
    cout<<"\n Enter Values of X & Y:";
    cin>>x>>y;
    swap(x,y);
    cout<<"\n In main() ";
    cout<<"\n Values X="<<x <<" and Y="<<y;
    cout<<"\n Addresses X="<<(unsigned)&x <<" Y
        ="<<(unsigned)&y;
    return 0;
}

void swap(int &a, int &b)
{
    int k; k=a; a=b; b=k;
    cout<<"\n In swap() ";
    cout<<"\n Values X="<<a <<" and Y="<<b;
    cout<<"\n Addresses X="<<(unsigned)&a <<" Y
        ="<<(unsigned)&b;
}
  
```

OUTPUT**Enter Values of X & Y :2 3****In swap ()****Values X=3 and Y=2****Addresses X=65524 Y =65522****In main ()****Values X=3 and Y=2****Addresses X=65524 Y =65522**

Explanation: In the above program, the function `swap ()` is invoked and actual arguments, variable `x` and `y`, are passed. The formal arguments of function `swap ()`, `a` and `b`, are reference variables of actual arguments `x` and `y`. The addresses of actual and formal arguments are same. Any change made in `a` and `b` leads to change in the values of `x` and `y`. To pass variables by reference, the function call is like call by value. In function declarator and function prototype declaration, the formal arguments are preceded by `&` operator. The reference type variable can be used in the same manner like other ordinary variables. The following points are noted concerning reference parameters:

- (a) A reference may not be null. It should always refer to a legal variable.
- (b) Once declared, a reference should not be altered to refer to another object.
- (c) A reference variable does not need any explicit address manipulation to access the actual value of the variable.

7.4 LVALUES AND RVALUES**7.4.1 Lvalues (Left Values)**

An expression that indicates the location is referred as **lvalue** and the expression that indicates the value is referred as **rvalue**. On left-hand side of the assignment operator (`=`), lvalues expression appears. Lvalues are quite often identifiers; whereas, on the right-hand side of the assignment operator (`=`), rvalue expression appears, which is a value to be stored at some address in memory. The set of rvalues is the subset of set of lvalues.

An lvalue is an object locator. It is an expression that points an object. An example of an lvalue expression is `*k` that results to a non-null pointer. A changeable lvalue is an identifier or expression that relates to an object that can be accessed and suitably modified in computer memory. A `const` pointer to a constant is not a changeable lvalue. A pointer to a constant can be altered (its de-referenced value cannot be altered). An lvalue could suitably stand on the left (the assignment side) of an assignment statement. Now, only changeable lvalue can legally stand on the left of an assignment statement. For example, suppose `x` and `y` are non-constant integer identifiers with appropriate allocated memory. Their lvalues are changeable. The following expressions are legal:

```
x = 1;
y = x + y are legal expressions.
```

7.4.2 Rvalues (Right Values)

The statement $x + y$ is not an lvalue, $x + y = z$ is invalid because the expression on the left is not related to a variable. Such expressions are often called *rvalues*.

7.5 RETURN BY REFERENCE

We have studied the reference variable and its functioning. A reference allows creating alias for the pre-existing variable. A reference can also be returned by the function. A function that returns reference variable is in fact an alias for referred variable. This technique of returning reference is used to establish cascade of member functions calls in operator overloading. Consider the following example:

The below given program illustrates return by reference.

7.7 Write a program to return a value by reference.

```
#include<iostream.h>
#include<conio.h>

int main()
{
    clrscr();
    int & min ( int &j, int &k);
    int a=18,b=11,c;
    c=min (a,b);
    cout<<"Minimum Value = "<<c;
    return 0;
}

int & min (int &j, int &k)
{
    if (k<j ) return k;
    else
        return j;
}
```

OUTPUT

Minimum Value = 11

Explanation: In the above program, the statement `int & min (int &j, int &k)` declares prototype of function `min()`. The '&' reference operator is used because the function returns reference to `int` and also receives arguments as reference. The function `min()` receives two integers as reference and returns minimum value out of two by reference.

7.8 Write a program to demonstrate return by reference.

```
#include<iostream.h>
#include<conio.h>
```

```

int & large (int & p, int & q);
int main()
{ clrscr();
  int l,k;
  cout<<"\n Enter values of l and k: ";
  cin>>l>>k;
  large (l,k)=120;
  cout<<" l= "<<l << " k="<<k;
  return 0;
}
int & large (int & p, int & q)
{
  if (p>q) return p;
  else return q;
}

```

OUTPUT

Enter values of l and k : 4 8

l= 4 k=120

Enter values of l and k : 9 2

l= 120 k=2

Explanation: In function `main()`, the statement `large (l, k) = 120` calls the function `large()`. It returns reference to the variable containing larger value and assigns the value 120 to it. The return type of function `large()` is `int '&'` (reference), it indicates that the call to function `large()` can be written on the left side of the assignment operator. Consequently, the statement `large(l,k) = 120` is legal and assigns 120 to the variable containing larger value.

7.6 RETURNING MORE VALUES BY REFERENCE

The `return()` statement has one big limitation that it can return only one value at a time. The `return()` statement is only used when values are passed by value to functions. Call by address and call by reference, accesses memory location directly. When user wants to return more than one value from function, he/she should pass values by address or by reference method. In C for returning more values, call by address is used. In C++ we have one additional method, that is call by reference. Consider the following program.

7.9 Write a program to return more than one value from function by reference.

```

#include<iostream.h>
#include<conio.h>

int main()
{
  int sq,cb,n;

```

```

void more (int &, int &, int);
cout<<"\n Enter a Number: ";
cin>>n;
more (sq,cb,n);
cout<<"\n Square ="<<sq;
cout<<"\n Cube ="<<cb;
return 0;
}
void more (int & s, int & c,int j)
{
    s=j*j;
    c=j*j*j;
}

```

OUTPUT

Enter a Number : 2

Square =4

Cube =8

Explanation: In the above program the function `more()` is used to perform square and cube of an integer passed. The three variables `sq`, `cb`, and `n` of integer type are declared. The number entered by the user is stored in variable `n`. The variables `sq`, `cb`, and `n` are passed to function `more()`.

In function `more()`, `s` and `j` are reference variables and `j` is a local variable of function `more()`. The variable `s` is reference variable of `sq` and variable `c` is reference variable of `cb`. The variables `s` and `sq`, and `c` and `cb` have the same memory locations. Thus, any value assigned to `s` and `c` can be accessed by `sq` and `cb`. The variable `s` is assigned the square of `j(n)` and variable `cb` is assigned with cube of `j(n)`.

7.7 DEFAULT ARGUMENTS

Usually, a function is called with all the arguments as declared in function prototype declaration and function definition. C++ compiler allows the programmer to assign default values in the function prototype declaration/function declarator of the function. When the function is called with less parameter or without parameters, the default values are used for the operations.

It is not allowed to assign default value to any variable, which is in between the variable list. Variables lacking default values are written first followed by the variables containing default values. This is so because C++ convention of storing variables on the stack is from right to left.

The default values can be specified in function prototype declaration or function declarator. Normally, the default values are placed in function prototype declaration. In case the function is defined before caller function, then there is no need to declare function prototype. Hence, the default arguments are placed in function declarator. The compiler checks for the default values in function prototype and function declarator and provides these values to the arguments that are omitted in function call.

The default arguments are useful while making a function call if we do not want to take effort for passing arguments that are always same. It is also useful when we update an old function by adding more arguments to it. Using default arguments, the function calls can continue to use previous arguments with the new arguments.

The following example declares the default arguments:

Example

```
int sum(int a, int b=10, int c=15, int d=20)
```

In this example function `sum()` has four arguments. They are `a`, `b`, `c`, and `d` of integer types. The variable `b`, `c`, and `d` are initialized with 10, 15, and 20, respectively. These values are used when the function `sum()` is called with fewer arguments.

7.10 Write a program to define function sum with default arguments.

```
#include<iostream.h>
#include<conio.h>

int main()
{
    clrscr();
    int sum (int a,int b=10,int c=15,int d=20 ); // Function
                                                    prototype

    int a=2;
    int b=3;
    int c=4;
    int d=5;
    cout<<"Sum=" <<sum(a,b,c,d) ;
    cout<<"\nSum=" <<sum(a,b,c) ;
    cout<<"\nSum=" <<sum(a,b) ;
    cout<<"\nSum=" <<sum(a) ;
    cout<<"\nSum=" <<sum(b,c,d) ;
    return 0;
}

sum (int j, int k, int l, int m)
{
    return (j+k+l+m);
}
```

OUTPUT

```
Sum=14
Sum=29
Sum=40
Sum=47
Sum=32
```

Explanation: In the above example, the prototype of variable `sum()` is declared. The function `sum()` has four arguments of integer types **a**, **b**, **c**, and **d**. The variables `b`, `c`, and `d` are initialized with default values 10, 15, and 20, respectively, in function prototype declaration. The variable "a" is not initialized. The function `sum()` is called five times. Each time the result has different values.

In the first `cout` statement, the function `sum()` is called with four arguments. In this function call, no default values are used, that is actual values of variables initialized are considered. Hence, the result is 14.

In the second `cout` statement the function `sum()` is called with three arguments (one less argument). In this function call actual values of starting three variables `a`, `b`, `c` and default value of forth variable "d" are considered. Hence, the result is 29.

In the third `cout` statement, the function `sum()` is called with two arguments (two arguments less). In this function call actual values of starting two variables `a`, `b` and default values of last two variables `c`, `d` is considered. Hence, the result is 40.

In the fourth `cout` statement the function `sum()` is called with one argument. In this function call, the actual value of variable `a` and default values of variables `b`, `c`, and `d`, 10, 15, and 20, respectively, are taken into account. The result displayed is 47.

In the last `cout` statement the function `sum()` is called with last three arguments. The actual values of `b`, `c`, and `d` variables are 3, 4, and 5 and last default value 20 is together taken into account. Hence the result is 32.

7.11 Write a program to place default values in function declarator. Execute the function with default values.

```
#include<iostream.h>
#include<conio.h>

sum (int j, int k=10, int l=15, int m=20)
{
    return (j+k+l+m);
}

int main()
{
    clrscr();
    int a=2;
    int b=3;
    int c=4;
    int d=5;
    cout<<"Sum=" <<sum(a,b,c,d);
    cout<<"\nSum=" <<sum(a,b,c);
    cout<<"\nSum=" <<sum(a,b);
    cout<<"\nSum=" <<sum(a);
    cout<<"\nSum=" <<sum(b,c,d);
    return 0;
}
```

OUTPUT

Sum=14
 Sum=29
 Sum=40
 Sum=47
 Sum=32

Explanation: In the above example, the prototype of variable `sum()` is declared. The function `sum()` has four arguments of integer types **a**, **b**, **c**, and **d**. The variable **b**, **c**, and **d** are initialized with default values 10, 15, and 20, respectively, in function prototype declaration. The variable "a" is not initialized. The function `sum()` is called five times. Each time the result sum has different values.

In the first `cout` statement, the function `sum()` is called with four arguments. In this function call no default values are used, that is actual values of variables initialized are considered. Hence, the result is 14.

In the second `cout` statement the function `sum()` is called with three arguments (one less argument). In this function call actual values of starting three variables **a**, **b**, **c** and default value of forth variable "d" are considered. Hence, the result is 29.

In the third `cout` statement, the function `sum()` is called with two arguments (two arguments less). In this function call actual values of starting two variables **a**, **b** and default values of last two variables **c**, **d** are considered. Hence, the result is 40.

In the fourth `cout` statement the function `sum()` is called with one argument. In this function call the actual value of variable **a** and default values of variables **b**, **c**, and **d** are 10, 15, and 20, respectively, are taken into account. The result displayed is 47.

In the last `cout` statement the function `sum()` is called with last three arguments. The actual values of **b**, **c**, and **d** variables are 3, 4, and 5 and last default value 20 is together taken into account. Hence the result is 32.

7.12 Write a program to find area of triangle by using default values and actual values.

The formula for area of a triangle is $\frac{1}{2} * \text{base} * \text{height}$.

```
#include<iostream.h>
#include<conio.h>
```

```
int main()
{
    clrscr();
    float area (int base=3,int height=5 ); // Function prototype
    int base=2;
    int height=7;
    cout<<"Area of Triangle: " << area(base,height);
    cout<<"\nArea of Triangle: " << area (base);
    cout<<"\nArea of Triangle: " <<area(height);
    return 0;
```



```

}
float area (int j, int k)
{ return (.5*j*k); }

```

OUTPUT

Area of Triangle : 7

Area of Triangle : 5

Area of Triangle : 17.5

Explanation: In the above program, the prototype of function `area()` is declared with two default values 3 and 5 for variables `base` and `height`, respectively. In function `main()` the variable `base` and `height` are defined and initialized with 2 and 7, respectively. In the first call of the function `area()` function is called with both the arguments, that is `base` and `height`. In this call no default arguments are used. In the second call the function `area()` is called with one argument (one less argument). In this call one actual and one default value are used. The third call is the same as second one.

7.8 const ARGUMENTS

The constant variable can be declared using `const` keyword. The `const` keyword makes variable value stable. The constant variable should be initialized while declaring.

Syntax:

```

(a) const <variable name> = <value>;
(b) <function name> (const <type>*<variable name>;)
(c) int const x // invalid
(d) int const x =5 // valid

```

In statement (a), the `const` modifier enables to assign an initial value to a variable that cannot be changed later by the program.

For example,

```
const age = 40;
```

Any attempt to change the contents of `const` variable `age` will produce a compiler error. Using pointer one can indirectly modify a `const` variable as per the following:

```
*(int *)&age = 45;
```

When the `const` variable is used with a pointer argument in a function's parameter list, the function cannot modify the variable that the pointer points to. For example,

```
int cube(const int *x, ...);
```

Here the `cube` function is prohibited from altering the integer variable.

7.13 Write a program to declare constant variable.

```

#include<iostream.h>
#include<conio.h>

```

```

int main()
{
    clrscr();
    int min (int const a=8, int b=20);
    int a=12,b=45;
    b=min (a);
    cout<<"\n a= "<<a <<" b= "<<b;
    return 0;
}
int min (int const j, int k)
{
    // j++; // can not modify a constant variable
    // k++; // valid because b is not constant
    cout<<"\n j= "<<j <<" k ="<<k;
    if (k<j ) return k;
    else return j;
}

```

OUTPUT

```

j= 12 k =20
a= 12 b= 12

```

Explanation: In the above program, the prototype of function `min()` declares two arguments. Out of two arguments the first argument `a` is constant and its default value is 8. The second variable is `b` and default value is 20. In function `main()` we can change the value of variable `a`, because in `main()` it is not declared as a constant. When function `min()` is invoked, the first argument `j` is a constant according to prototype of function. Any attempt made to change the value of variable `j` will cause an error message “cannot modify a constant variable.”

7.14 Write a program to change the value of a constant variable using pointer.

```

#include<iostream.h>
#include<conio.h>
#include<process.h>

int main()
{
    clrscr();
    int const x=1;
    // x++; invalid operation
    // cin>>x; possible, but no change in constant value
    ++(*(int *)&x);
    cout<<"x = "<<x <<"\t\t Address: "<<(unsigned)&x;
    cout<<"\n*(&x) = "<<*(&x)<<"\t\t Address: " <<(unsigned)&x;
    return 0;
}

```

```
}

```

OUTPUT

```
x = 1 Address : 65524
*(&x) = 2 Address : 65524
```

Explanation: In the above program, the variable `x` is a constant integer variable. In C it is possible to change the value of a constant variable using `scanf()` statement, that is run time assignment. In C++, though compiler accepts this statement, the value is not changed. We can change the value of constant variable using a pointer. The statement `++(* (int *) &x);` points to the address of the variable `x` and increase is done in the value stored at the location, that is indirectly to variable `x`. Though it is possible to change the value of a variable using pointer, the following statements are invalid.

```
int const x=1;
int const *p=&x;
++*p;
```

It is not possible to change the value of a variable using pointer `*p`. From the above program we learned that *the value of a constant could be changed using pointer*.

7.9 INPUTTING DEFAULT ARGUMENTS

Default arguments are used when function is invoked with fewer arguments than declared. The default values are placed in function prototype or function declarator. It is also possible to directly invoke user-defined function in function prototype or function declarator. The return value of function is considered as default value. The following program illustrates this point:

7.15 Write a program to enter default values using function.

```
#include<iostream.h>
#include<conio.h>

int main()
{
    clrscr();
    int input(void);
    void display (int = input());
    display(20); // first call with argument
    display(); // second call with default argument
    return 0;
}

void display (int j)
{
```

```
    cout<<"\nInteger is: "<<j;
}
input()
{
    int k;
    cout<<"\n Enter an integer: ";
    cin>>k;
    return k;
}
```

OUTPUT

Integer is : 20

Enter an integer : 12

Integer is : 12

Explanation: In the above program, user-defined functions `display()` and `input()` are declared and defined. The `display()` function is used to display passed integer on the screen. The `input()` function when executed asks for an integer. The function `input()` is used as default value for function `display()`. When a function `display()` is invoked without argument, the `input()` function assigned as default argument in function prototype is executed. User enters a number. The entered number is then passed on to `display()` function and displayed on the screen.

7.10 INLINE FUNCTIONS

One of the prime factors behind using function is that code duplication in program is avoided and memory space is saved. When a function is defined and invoked, one set of instruction is created in the memory of the system. At each call, the control passes to the subroutine at a specified address in the memory. The CPU stores the memory address of instruction following the function calls in to the stack and also pushes the arguments in the stack area when call is made to the function. The compiler runs the function, stores the return values in a memory location or register, and when execution completes, the control returns to the calling function. After this, execution resumes immediately to the next line following the function call statement. If the same function is called several times, each time the control is passed to the function, that is the compiler uses the same set of instructions at each call. Due to this passing of control in between caller and callee functions, execution speed of program decreases. By preventing the repetitive calls in a program, execution speed can be increased.

C++ provides a mechanism called `inline` function. When a function is declared as `inline`, the compiler copies the code of the function in the calling function, that is function body is inserted in place of function call during compilation. Passing of control between caller and callee functions is avoided. If the function is very large, in such a case `inline` function is not used because the compiler copies the contents in the called function that reduces the program execution speed. The `inline` function is mostly useful when calling function is small. It is advisable to use the `inline` function for only small functions.

Inline mechanism increases execution and performance in terms of speed. The overhead of repetitive function calls and returning values are removed. On the other hand, the program using inline functions needs more memory space, since the inline function are copied at every point where the function is invoked.

For defining an inline function, the `inline` keyword is preceded by the function name. The inline functions are declared as follows:

```
inline function - name
{
    statement1;
    statement2; // Function body
}
```

```
Example:
inline float square (float k)
{
    return (k*k);
}
```

The above example can be executed as follows:

```
j = square(2.5);
k = square(1.1 + 1.4);
```

After the execution of above statements, the values of `j` and `k` will be 6.25. The `inline` keyword just makes an appeal to the compiler. The compiler may neglect this request if the function defined is too big in size or too convoluted. In such a case the function is treated as normal function.

Following are few situations where inline functions may not work:

- (1) The function should not be recursive.
- (2) Function should not contain static variables.
- (3) Function containing control structure statements such as `switch`, `if`, `for` loop, etc.
- (4) The function `main()` cannot be used as inline.

The inline functions are similar to macros of C. The main limitation of macros is that they are not functions, and errors are not checked at the time of compilation. The function offers better type testing and does not contain limitations as present in macros. Consider the following example:

7.16 Write a program to calculate square using inline functions and macros.

```
#include<iostream.h>
#include<conio.h>
#define SQUARE(v) v * v

inline float square (float j)
{
```




```

    return (j*j);
}
int main()
{
    clrscr();
    int p=3, q=3,r,s;
    r=SQUARE(++p);
    s=square(++q);
    cout<<" r= "<<r<<"\n"<<" s= "<<s;
    return 0;
}

```

OUTPUT

r= 25

s= 16

Explanation: In the above program, the function `square()` is declared as an inline function. The macro `square()` expanded into `r = ++v * ++v`. The variable `p` is incremented only once but in macros expansion is incremented twice. Therefore, it gives wrong result.

7.17 Write a program to define an inline function and obtain the result.

```

#include<iostream.h>
#include<conio.h>

int main()
{
    clrscr();
    int sum (int,int,int,int ); // Function prototype
    cout<<"Sum=" <<sum(5,4,3,1);
    return 0;
}

inline sum (int j, int k, int l, int m)
{
    return (j+k+l+m);
}

```

OUTPUT

Sum = 13

Explanation: In the above program, the function `sum()` calculates the sum of given integers. The function is defined as inline. The compiler copies the code of the function in the calling function and executes it.

7.18 Write a program to define function cube() as inline for calculating cube.

```
#include<iostream.h>
#include<conio.h>

int main()
{
    clrscr();
    int cube (int);
    int j,k,v=5;
    j = cube (3) ;
    k = cube (v);
    cout<<"\n Cube of j="<<j;
    cout<<"\n Cube of k="<<k;
    return 0;
}

inline int cube (int h)
{
    return (h*h*h);
}
```

OUTPUT**Cube of j=27****Cube of k=125**

Explanation: In the above program, the function cube () is declared as inline. The function cube () calculates cube of passed number. The function is declared as inline, that is the statements of function cube () are inserted at the point of function call.

7.11 FUNCTION OVERLOADING

It is possible in C++ to use the same function name for number of times and for different intentions. Defining multiple functions with same name is known as function overloading or function polymorphism. Polymorphism means one function having many forms. The overloaded function must be different in their argument list and with different data types. The following are examples of overloaded functions. All the functions defined should be equivalent to their prototypes.

```
int  sqr(int);
float  sqr(float);
long  sqr(long);
```

7.19 Write a program to calculate square of an integer and float number. Define function sqr(). Use function-overloading concept.

```
#include<iostream.h>
#include<conio.h>
```

```

int sqr (int);
float sqr (float);

int main()
{
    clrscr();
    int a=15;
    float b=2.5;
    cout<<"Square = "<<sqr(a) <<"\n";
    cout<<"Square = "<< sqr(b) <<"\n";
    return 0 ;
}

int sqr (int s)
{
    return (s*s);
}

float sqr (float j)
{
    return (j*j);
}

```

OUTPUT

Square = 225

Square =6.25

Explanation: In the above function, `sqr()` is overloaded for integer and `float`. In the first call of the function `sqr()`, an integer 15 is passed. The compiler executes integer version of the function and returns result 225. In the second call a `float` value 2.5 is passed to function `sqr()`. In this call, the compiler executed the `float` version of the function and returns the result 6.25. The selection of which function to execute is decided at run time by the compiler according to the data type of variable passed.

7.20 Write a program to find the area of rectangle, triangle and surface area of sphere. Use function overloading.

```

#include<iostream.h>
#include<conio.h>
#define pi 3.142857142857142857
int calcarea(int length,int breadth);
float calcarea(double base,double height);
float calcarea(double radius);

int main()
{
    int area1;
    float area2;

```



```

double area3;
area1=calcare(10,20);
area2=calcare(4.5,2.1);
area3=calcare(3.12145);
clrscr();
cout<<"Area of rectangle is: "<<area1<<endl;
cout<<"Area of triangle is: "<<area2<<endl;
cout<<"Surface Area of sphere is: "<<area3<<endl;
return 0;
}
int calcare(int length,int breadth)
{
    return (length*breadth);
}
float calcare(double base,double height)
{
    return ((0.5)*base*height);
}
float calcare(double radius)
{
    return (4*pi*radius*radius);
}

```

OUTPUT

Area of rectangle is : 200

Area of triangle is : 4.725

Surface Area of sphere is : 122.489087

Explanation: In the above program, function **calcare()** is overloaded. It is used for finding the area of a rectangle, triangle, and sphere. Explanation is same as detailed in previous problem.

7.12 PRINCIPLES OF FUNCTION OVERLOADING

- (1) If two functions have the similar type and number of arguments (data type), the function cannot be overloaded. The return type may be similar or void, but argument data type or number of arguments must be different. For example,

(a) `sum(int, int, int);`
`sum(int, int);`

Here, the above function can be overloaded. Though the data type of arguments in both the functions are similar, number of arguments are different.

(b) `sum(int, int, int);`
`sum(float, float, float);`

In the above example, number of arguments in both the functions are same, but data types are different. Hence, the above function can be overloaded.

- (2) Passing constant values directly instead of variables also results in ambiguity. For example,
- ```
int sum(int, int);
float sum(float, float, float);
```
- Here, `sum()` is an overloaded function for integer and float values. Values are passed as follows:

```
sum(2, 3);
sum(1.1, 2.3, 4.3);
```

The compiler will flag an error because the compiler cannot distinguish between these two functions. Here, internal conversion of float to int is possible. Hence, in both the above calls integer version of function `sum()` is executed. To overcome this problem, the user needs to do the following things.

- (1) Declare prototypes of all the overloaded functions before function `main()`.
- (2) Pass argument using variables as follows:

```
sum(a,b); // a and b are integer variables
sum(e,r,t,y); // e,r,t, and y are float variables
```

Refer program number 7.22 for confirmation. Also try this program with direct values and by declaring function prototype inside `main()`.

- (3) The compiler attempts to find an accurate function definition that matches in types and number of arguments and invokes that function. The arguments passed are checked with all declared function. If matching function is found then that function gets executed.
- (4) If there are no accurate match found, compiler makes the implicit conversion of actual argument. For example, `char` is converted to `int` and `float` is converted to `double`. If all the above steps have failed, then compiler performs user-built functions.

The following example illustrates this point:

**7.21 Write a program to overload a function and create the situation that the compiler does integral conversion.**

```
#include<iostream.h>
#include<conio.h>

int sqr (int);
float sqr (double);
int main()
{
 clrscr();
 int a=15;
 float b=3.5;
 cout<<"Square = "<<sqr('A') <<"\n";
 cout<<"Square = "<< sqr(b) <<"\n";
 return 0 ;
}

int sqr (int s) { return (s*s); }
float sqr (double j) { return (j*j); }
```

**OUTPUT**

**Square = 4225**  
**Square = 12.25**

**Explanation:** In the above program, the function square is overloaded. The first prototype of function `sqr(int)` is proposed for integer variable and second for `double` variable. In function `main()` the "A" character data type value is passed to the function `sqr()`. The compiler invokes the integer version of the function `sqr()` because the `char` data type is compatible with integer. In the second call, a `float` value is passed to the function `sqr()`. This time the compiler invokes the `double` data type version of function `sqr()`.

When accurate match is not found the compiler makes internal conversion as seen in the above example. C++ automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function.

- (5) If internal conversion fails, user-defined conversion is carried out with implicit conversion and integral promotion. The user-defined conversion is used with class objects.
- (6) Sometimes while making internal conversion, ambiguity is created if one data type is compatible with two or more data types. If such a situation occurs, the compiler displays an error message.

Consider the following example. Following are two versions of `sqr()` function, one is for long data type and other for double data type.

```
long sqr(long);
double sqr(double);
sqr(10); // function call
```

If an integer value is passed to the function `sqr()`, the compiler will have confusion of which version is to be executed and will result in an error message "**Ambiguity between 'sqr(long)' and 'sqr(double)'**".

- (7) The program has two versions of function, that is one for float and second for double data type. In such a situation if we pass a float number, the double version of function is selected for execution. But the same is not applicable with integer and long integer.

### **7.22 Write a program to define overloaded function `add()` for integer and float and perform the addition.**

```
#include<iostream.h>
#include<conio.h>

int add (int, int, int);
float add (float, float, float);
int main()
{
 clrscr();
```

```

float fa,fb,fc,fd;
int ia,ib,ic,id;
cout<<"\n Enter values integer values for ia,ib and ic: ";
cin>>ia >>ib >>ic;
cout<<"\n Enter float values for fa,fb and fc: ";
cin>>fa >>fb >>fc;
id=add (ia,ib,ic);
cout<<"\n Addition: " <<id;
fd= add (fa,fb,fc);
cout<<"\n Addition: " <<fd;
return 0;
}
add (int j, int k, int l)
{
 return (j+k+l);
}
float add (float a, float b, float c)
{ return (a+b+c); }

```

## OUTPUT

**Enter values integer values for ia,ib and ic : 1 2 4**

**Enter float values for fa,fb and fc : 2.2 3.1 4.5**

**Addition : 7**

**Addition : 9.8**

**Explanation:** In the above program, two versions of function `add()` are declared. One for integer values and the other for float values. Three integers and float values are entered through the keyboard. According to the data type, the compiler executes appropriate function. The prototype of functions should be written before `main()` function. If we write the prototype inside the `main()` function, the float version of the function will not be executed.

### 7.23 Write a program to define overloaded function. Invoke the overloaded function through another overloaded function.

```

#include<iostream.h>
#include<conio.h>

int add (void);
float add (float, float, float);
float fa,fb,fc,fd;
int id;
int main()
{
 clrscr();

```

```

 cout<<"\n Enter float values for fa,fb and fc: ";
 cin>>fa >>fb >>fc;
 id=add();
 cout<<"\n Addition: " <<id;
 fd= add (fa,fb,fc);
 cout<<"\n Addition: " <<fd;
 return 0;
}
add ()
{
 int x;
 x=add(fa,fb,fc);
 return (x);
}
float add (float a, float b, float c)
{
 return (a+b+c);
}

```

## OUTPUT

**Enter float values for fa,fb and fc : 5.5 2.4 8.2**

**Addition : 16**

**Addition : 16.1**

**Explanation:** In the above program, the function `add()` is overloaded. The `add(void)` function does not require any argument but returns integer. The `float add()` function requires three arguments of `float` type. Here, the arguments are `fa`, `fb`, `fc`, `fd`, and `id` and are declared as global so that any function can call them. After entering three float values, the `add(void)` function is called. The `add(void)` function calls the `float` type `add()`, the `float` type `add()` function calculates addition and returns results to function `add(void)`. The `add(void)` function then returns addition as an integer. In the second call, the compiler calls the `float add()` function; this function returns addition of float values. The output of the program is shown above.

## 7.13 PRECAUTIONS WITH FUNCTION OVERLOADING

Function overloading is a powerful feature of C++. But, this facility should not be overused. Otherwise it becomes an additional overhead in terms of readability and maintenance. Following precautions must be taken:

- (1) Only those functions that basically do the same task, on different sets of data, should be overloaded. The overloading of function with identical name but for different purposes should be avoided.
- (2) In function overloading, more than one function has to be actually defined and each of these occupy memory.



- (3) Instead of function overloading, using default arguments may make more sense and fewer overheads.
- (4) Declare function prototypes before `main()` and pass variables instead of passing constant directly. This will avoid ambiguity that frequently occurs while overloading functions.

## 7.14 RECURSION

In programming there might be a situation where a function needs to invoke itself. The C++ language supports recursive feature, that is a function calls itself repetitively. In recursion a function calls itself and the control goes to the same function and it executes repeatedly until some condition is satisfied. In this type of recursive calls a function starts with a new value every time.

### 7.14.1 Rules for Recursive Function

- (1) In recursion, it is essential to call a function by itself; otherwise recursion would not take place.
- (2) Only the user-defined function can be involved in the recursion. Library function cannot involve in recursion because their source code cannot be viewed.
- (3) A recursive function can be invoked by itself or by other function. It saves return address with the intention to return at proper location when return to a calling statement is made.

The last-in-first-out nature of recursion indicates that stack data structure can be used to implement it.

- (4) Recursion is turning out to be increasingly important in non-numeric applications and symbolic manipulations.
- (5) To stop the recursive function, it is necessary to base the recursion on test condition, and proper terminating statement such as `exit()` or `return()` must be written using the `if()` statement.

#### 7.24 Program to find the factorial of a entered number.

```
#include<conio.h>
#include<iostream.h>

int main()
{
 unsigned long int fact(int);
 int f,x;
 clrscr();
 cout<<"\nEnter a Number:";
 cin>>x;
 f=fact(x);
 cout<<"\nFactorial of " <<x<<" is "<<f;
 return 0;
}

unsigned long int fact(int a)
{
 unsigned long factorial;
```

```
 if(a==1)
 return 1;
 else
 factorial=a*fact(a-1);
 return factorial;
}
```

## OUTPUT

**Enter a Number:6**  
**Factorial of 6 is 720**

### 7.25 Program to calculate the gcd of two numbers.

```
#include<conio.h>
#include<iostream.h>
int gcd(int,int);

int main()
{
 clrscr();
 int a,b,z;
 cout<<"\nEnter two numbers:";
 cin>>a>>b;
 z=gcd(a,b);
 cout<<z;
 return 0;
}

int gcd(int a,int b)
{
 if(a!=b)
 {
 if(a>b)
 gcd(a-b,b);
 else
 gcd(a,b-a);
 }
 else
 return(a);
}
```

{coderipe}

## OUTPUT

**Enter two numbers:45 60**  
**GCD of 45 and 60 is 15**

## 7.15 LIBRARY FUNCTIONS

### 7.15.1 Ceil, ceil, floor, floorl

The functions `ceil` and `ceil` round up the given float number, whereas the functions `floor` and `floorl` round down the float number. They are defined in `math.h` header file. Their declarations are as given below.

| Declaration                                       |
|---------------------------------------------------|
| <code>Double ceil(double n);</code>               |
| <code>Double floor(double n);</code>              |
| <code>Long double ceil(long double (n));</code>   |
| <code>Long double floorl(long double (n));</code> |

The following program illustrates the working of these functions.

#### 7.26 Write a program to round down and up the given float number.

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

int main(void)
{
 clrscr();
 float num = 3.12;
 float d, u;
 d = floor(num);
 u = ceil(num);
 cout<<"\n Original number is: "<<num;
 cout<<"\n Number rounded up is: "<<u;
 cout<<"\n Number rounded down is: "<<d;
 return 0;
}
```

#### OUTPUT

**Original number is : 3.12**  
**Number rounded up is : 4**  
**Number rounded down is : 3**

**Explanation:** In the above program, the float variable `num` has a value `3.12`. The `floor()` function converts it to the nearest and small integer number than variable `num` and returns it to variable `d`, whereas the `ceil()` function converts the float number to the nearest and greater number than `num`. The output of the program is shown above.

### 7.15.2 modf and modfl

The function `modf` breaks double into integer and fraction elements, and the function `modfl` breaks long double into integer and fraction elements. These functions return the fractional elements of a given number. They are declared as given below.

|                                                                                                      |
|------------------------------------------------------------------------------------------------------|
| Declaration                                                                                          |
| <pre>double modf(double n, double *ip); long double modfl(long double (n), long double *(ip));</pre> |

#### 7.27 Write a program to separate double number into integer and fractional parts.

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

int main()
{
 clrscr();
 double f, i;
 double num = 211.57;
 f = modf(num, &i);
 cout<<"\n The Complete Number: "<<num;
 cout<<"\n The Integer elements: "<<i;
 cout<<"\n Fractional Elements: "<<f;
 return 0;
}
```

#### OUTPUT

**The Complete Number : 211.57**

**The Integer elements : 211**

**Fractional Elements : 0.57**

**Explanation:** In the above program, the `modf()` function separates a double number into separate integer and fractional parts. The second argument of the function is passed by reference that after execution holds the integer part of the number.

### 7.15.3 abs, fabs, and labs

The function `abs()` returns the absolute value of an integer. The `fabs()` returns the absolute value of a floating-point number, and `labs()` returns the absolute value of a long number.

|                                                                                              |
|----------------------------------------------------------------------------------------------|
| Declaration:<br><pre>int abs(int n); double fabs(double n); long int labs(long int n);</pre> |
|----------------------------------------------------------------------------------------------|

**abs :** If **abs** is invoked using *STDLIB.H* , it is executed as a macro that expands to inline code. If we need to use the original **abs** function then undefine the macro using statement **#undef abs** in the program, after the **#include<stdlib.h>**.

The following program illustrates the working of these functions.

### 7.28 Write a program to display the absolute number of an integer, float, and long integer.

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

int main()
{
 clrscr();
 int num = -1874;
 cout<<"\n Number: "<<num <<"\t\t Absolute value: "
 <<abs (num) ;

 {
 float num = -111.11;
 cout<<"\n Number: "<<num <<"\t Absolute value: "<<fabs (num) ;
 }
 long ans;
 long n = -12252111L;
 ans= labs (n) ;
 cout<<"\n Number: "<<n <<"\t Absolute value: "<<ans;
 return 0;
}
```

### OUTPUT

```
Number : -1874 Absolute value : 1874
Number : -111.110001 Absolute value : 111.110001
Number : -12252111 Absolute value : 12252111
```

**Explanation:** In the above program, the function **abs ( )** returns the absolute value of a negative number **-1874** . The **fabs ( )** function returns the absolute value of a floating-point number. The **labs ( )** function returns the absolute value of long number. In the statement **long n = -12252111L**, the **L** tells the compiler that the number is of long data type and provides storage equal to and as required for long data type.

### 7.15.4 norm

This function is defined in **complex.h** header file and it is used to calculate the square of the absolute value.

Syntax: `double norm(complex n) ;`

### 7.29 Write a program to calculate the square of the complex number using norm() function.

```
#include<iostream.h>
#include<complex.h>
#include<conio.h>

int main()
{
 clrscr();
 double x=-12.5;
 cout<<norm(x);
 return 0;
}
```

#### OUTPUT

6.25

**Explanation:** In the above program the norm() function calculates the square of complex negative number.

### 7.15.5 complex(), real(), imag(), and conj()

**complex():** This function is defined in complex.h header file and it creates complex numbers from given real and imaginary parts. The imaginary part is supposed to be 0 if not given. Complex is the constructor for C++ class complex. This function returns the complex number with the given real and imaginary parts.

**real():** Returns real part of the complex number.

**imag():** Returns the imaginary part of the complex number.

**conj():** Returns complex conjugate of a complex number.

```
Syntax: complex complex(double real, double imag);
Syntax: double real(complex x);
Syntax: double imag(complex x);
Syntax: double conj(complex x);
```

### 7.30 Write a program to return real, imaginary part and conjugate number.

```
#include<conio.h>
#include<iostream.h>
#include<complex.h>

int main(void)
{
 double a = 4.5, b = 7.4;
 clrscr();
```

```

 complex c = complex(a,b);
 cout<< "c = "<< c << "\n";
 cout<< " and imaginary real part = " << imag(c) << "\n";
 cout<< "c has complex conjugate = " << conj(c) << " \n";
 return 0;
}

```

## OUTPUT

**C = (4.5, 7.4)**  
**and imaginary real part = 7.4**  
**c has complex conjugate = (4.5, -7.4)**

**Explanation:** In the above program variable a and b of double data type are declared and initialized with 4.5 and 7.4. By applying the `complex()`, `imag()` and `conj()` functions results are obtained. The results are shown in the output.

## 7.16 MORE PROGRAMS

**7.31 Write a program to convert a float value to integer and char. Invoke overloaded function `show()` as per the data type.**

```

#include<iostream.h>
#include<conio.h>
void show (int);
void show (float);
void show (char);

int main()
{
 clrscr();
 float x;
 int i;
 char c;
 for (x=65;x<=70;x+=0.5)
 {
 show(x);
 i=(int)x;
 show(i);
 c=(char)x;
 show(c);
 cout<<endl;
 }
 return 0;
}

```

```

void show (int j) {cout<<"\tint = "<<j <<"\t";}
void show (float k) {cout<<' \t'<<" float = " <<k;}
void show (char c) {cout<<' \t' <<" char = " <<c;}

```

## OUTPUT

```

float = 65 int = 65 char = A
float = 65.5 int = 65 char = A
float = 66 int = 66 char = B
float = 66.5 int = 66 char = B
float = 67 int = 67 char = C
float = 67.5 int = 67 char = C
float = 68 int = 68 char = D
float = 68.5 int = 68 char = D
float = 69 int = 69 char = E
float = 69.5 int = 69 char = E
float = 70 int = 70 char = F

```

**Explanation:** In the above program, the function `show()` is overloaded for `float`, `int`, and `char` data types, respectively. In for loop, conversion from `float` to `int` and `float` to `char` data type is carried out and converted values are assigned to variables `i` and `c`, respectively. After each conversion `show()` function is invoked. The compiler invokes the appropriate function that exactly matches the data type argument. Thus, output displays `float`, `int`, and corresponding *ASCII* character.

### 7.32 Write a program to use function as default arguments.

```

#include<iostream.h>
#include<conio.h>

int main()
{
 clrscr();
 int in();
 int sum (int a,int b=in(),int c=in(),
 int d=in()); // Function prototype

 int a=2;
 int b=3;
 int c=4;
 int d=5;
 cout<<"Sum=" <<sum(a,b,c,d) ;
 cout<<"\nSum=" <<sum(a,b,c) ;
 cout<<"\nSum=" <<sum(a,b) ;
 cout<<"\nSum=" <<sum(a) ;
 return 0;
}

```



```

sum (int j, int k, int l, int m)
{
 return (j+k+l+m);
}
in()
{
 static int j=1;
 int k;
 cout<<"\n (call "<<j++<<") Enter integer number: ";
 cin>> k;
 return (k);
}

```

## OUTPUT

```

Sum=14
(call 1) Enter integer number : 2
Sum=11
(call 2) Enter integer number : 3
(call 3) Enter integer number : 4
Sum=12
(call 4) Enter integer number : 5
(call 5) Enter integer number : 4
(call 6) Enter integer number : 5
Sum=16

```

**Explanation:** In the above program, the function `in()` is used to read an integer value through the keyboard. The function `in()` is used in the prototype of function `sum()` as a default argument. Thus, when `sum()` function is called with less arguments, the function `in()` is executed and user can enter a value to pass in the `sum()` function. The execution of `in()` function depends upon the number of arguments passed to the function `sum()`. In this program, the function `sum()` is invoked with decreasing arguments from 4 to 1.

First time the function `sum()` is called with four arguments. Hence, there is no need of default argument and function `in()` will not be invoked. In second call, three arguments are passed, that is with one less argument. This time `sum()` needs one default argument and `in()` function supplies this value. Thus, in the above successive calls the function `in()` is called based on passed arguments.

### 7.33 Write a program to use global variable as default argument.

```

#include<iostream.h>
#include<conio.h>
int v=2;

int main()

```

```

{
 clrscr();
 int show (int d=v+5);
 int a=2;
 cout<<"\n a++ =" <<show(a);
 cout<<"\n a++ =" <<show();
 v++;
 cout<<"\n a++ =" <<show();
 return 0;
}
show (int j)
{
 j++;
 return (j);
}

```

### OUTPUT

```

a++ =3
a++ =8
a++ =9

```

**Explanation:** In the above program, function `show()` is used to increase the value passed to it. The function `show()` returns increased value. The prototype of function `show()` contains default arguments `(v + 5)`, that is value of variable `v` is added to 5 and the result is used as a default argument for the function `show()`. In the first call, the function `show()` is called with the single argument and it returns value 3. In second call, `show()` is called without argument. This time function `show()` takes default argument, that is `v + 5 = 7` and returns the result 8. The third time the function `show()` is again called without argument. This time it returns the result 9 because the global variable `v` is incremented by 1. Thus, the use of global variable as a default argument changes the default value in different calls. The use of local variable as a default argument is not allowed. If such attempt is made, the compiler shows an error message “*Default expression may not use local variables.*”

### 7.34 Write a program to use function as an argument for another function.

```

#include<iostream.h>
#include<conio.h>
int v=2;

int main()
{
 clrscr();
 int show (int d=v+5);
 cout<<"\n ++ =" <<show(show());

```

```
 return 0;
}
show (int j)
{
 j++;
 return (j);
}
```

## OUTPUT

**++=9**

**Explanation:** In the above program the function `show()` is used as an argument. When function is used as an argument, the argument function is called first and its return value is used as an argument for the first function. In the first call the function uses the default value 7 and it increments by one. The return value 8 is again passed to the function `show()`. The return value again increases by one and finally the `show()` function returns the value 9.

### 7.35 Write a program to overload a function `display()`. Print the integer, double and string using overloaded `show()` function.

```
#include<conio.h>
#include<iostream.h>
void display (int d) { cout<<endl<<" d= "<<d; }
void display (double d) { cout<<endl<<" d= "<<d; }
void display (char *d) { cout<<endl<<" d= "<<d; }

int main()
{
 clrscr();
 display (50);
 display(3.14521);
 display("Hello");
 return 0;
}
```

## OUTPUT

**d= 50**

**d= 3.14521**

**d= Hello**

**Explanation:** In the above program the function `display()` is overloaded for displaying integer, double and string. In function `main()`, integer, double and string are passed to function `display()`. The compiler executed appropriate function `display()` and displays the passed value on the screen.

**7.36 Write a program to declare a function with default arguments. Execute the function without argument.**

```
#include<conio.h>
#include<iostream.h>
void display (char *s="Hearty Welcome\n")
{
 cout<<endl<<s;
}
int main()
{
 clrscr();
 display("Hello");
 display();
 return 0;
}
```

**OUTPUT**

**Hello**  
**Hearty Welcome**

**Explanation:** In the above program character pointer *s* is initialized with string “ Hearty Welcome” and it is declared as default arguments for function `display()`. The default argument is used when the function is invoked without argument. Here, in the first call a string “Hello” is passed and it is displayed on the screen by the function `display()`. In second call, the function `display()` is called without argument. In this call the default argument is used and the string displayed will be “ Hearty Welcome”.

**7.37 Write a program to draw line. Create line function with default arguments.**

```
#include<conio.h>
#include<iostream.h>
int main()
{
 void line (char = '\-', int = 50);
 clrscr();
 line(); // without arguments
 line ('\+', 50);
 line ('\/', 50);
 line ('\S', 50);
 line(); // without arguments
 return 0;
}
```



```
 cout<<"\nDivision of given numbers is "<<e;
 cout<<"\nModulus of given numbers is "<<f;
 return(0);
}
float add (float x,float y)
{
 return(x+y);
}
float sub(float x,float y)
{
 return(x-y);
}
float mul (float x,float y)
{
 return(x*y);
}
float div(float x,float y)
{
 return(x/y);
}
int mod(int p,int q)
{
 return(p%q);
}
```

## OUTPUT

**Enter numbers 5 2**

**Addition of given numbers is 7**

**Subtraction of given numbers is 3**

**Multiplication of given numbers is 10**

**Division of given numbers is 2.5**

**Modulus of given numbers is 1**

### 7.39 Program to display function show() with default parameters

```
#include<iostream.h>
#include<conio.h>
void show(int=1,float=7.5,long=9);
void main()
{
 clrscr();
 show();
}
```

```
 show(5);
 show(3,2.4);
 show(1,7.7,31);
}
void show(int first,float second,long third)
{
 cout<<"\nFirst = "<<first;
 cout<<" , second = "<<second;
 cout<<" , third = "<<third;
}
```

## OUTPUT

```
First = 1,second = 7.5,third = 9
First = 5,second = 7.5,third = 9
First = 3, second = 2.4, third = 9
First = 1, second = 7.7, third = 3
```

### 7.40 Program to find addition and multiplication of digits of any number.

```
#include<iostream.h>
#include<conio.h>

int main()
{
 int n,p,q;
 int sum(int);
 int mul(int);
 clrscr();
 cout<<"Enter any number ";
 cin>>n;
 p= sum(n);
 q= mul(n);
 cout<<"Sum of digits "<<p;
 cout<<"\nMultiplication of digits "<<q;
 return 0;
}

int sum(int s)
{
 int a,sum=0,rem;
 for(a=s;a>0;a/=10)
 {
 rem=a%10;
 sum=sum+rem;
 }
}
```

```
 }
 return(sum);
}
int mul(int m)
{
 int e,res=1,rem;
 for(e=m;e>0;e/=10)
 {
 rem=e%10;
 res*=rem;
 }
 return (res);
}
```

## OUTPUT

**Enter any number 123**

**Sum of digits 6**

**Multiplication of digits 6**

### 7.41 Program to convert binary number to decimal.

```
#include<iostream.h>
#include<conio.h>

int main()
{
 int b;
 clrscr();
 cout<<"Enter any number in binary:";
 cin>>b;
 void decimal(int);
 decimal(b);
 return 0;
}

void decimal(int n)
{
 int k,rem,d,dec=0,a=1;
 for(k=n;k>0;k/=10)
 {
 rem=k%10;
 d=rem*a;
 dec+=d;
 a*=2;
 }
}
```





```
 cout<<" Binary number= " <<n<<"\nDecimal number= "<<dec;
}
```

## OUTPUT

**Enter any number in binary:1111**  
**Binary number= 1111**  
**Decimal number= 15**

### 7.42 Program to overload function addition of integers and floats.

```
#include<iostream.h>
#include<conio.h>
int add (int,int);
float add(float,float,float);

int main()
{
 int a,b,i;
 float p,q,r,f;
 clrscr();
 cout<<"Enter integers: a&b ";
 cin>>a>>b;
 cout<<"Enter floats: p,q & r ";
 cin>>p>>q>>r;
 i=add(a,b);
 f=add(p,q,r);
 cout<<"\nAddition of integers= "<<i;
 cout<<"\nAddition of floats= "<<f;
 return 0;
}

int add(int x,int y)
{
 return(x+y);
}

float add(float m,float n,float o)
{
 return(m+n+o);
}
```

## OUTPUT

**Enter integers : a&b 2 7**  
**Enter floats: p,q & r 2.4 5.6 7.6**  
**Addition of integers= 9**  
**Addition of floats= 15.6**

## SUMMARY

- (1) Like “C” language, the programs written in “C++” language highly depend on functions. The “C++” program is nothing but a combination of one or more functions.
- (2) A function prototype declaration consists of function’s return type, name, and arguments list. When the programmer defines that function, the definition must be the same as its prototype declaration.
- (3) In C++ it is possible to pass arguments by value or by reference. C++ reference type is declared with & operator and it is nearly identical but not exactly the same as pointer types. They declare aliases for objects variables and allow the programmer to pass arguments by reference to functions.
- (4) An *lvalue* is an object locator and an expression that indicates an object.
- (5) The statement  $x + y$  is not an *lvalue*;  $x + y = z$  is invalid because the expression on the left is not related to a variable. Such expressions are often called *rvalues*.
- (6) C++ allows the programmer to assign default values in the function prototype declaration of the function. When the function is called with lesser parameters, then the default values are used for the operations.
- (7) The constant variable can be declared using `const` keyword. The `const` keyword makes variable value stable. The constant variable should be initialized at the time of declaration.
- (8) C++ provides a mechanism called *inline* function. When a function is declared as *inline*, the compiler copies the code of the function in the calling function.
- (9) C++ makes it possible for the programmer to use the same function name for various times for different intentions. This is called function overloading or function polymorphism.
- (10) The function `modf()` breaks double into integer and fraction elements, and the function `modfl()` breaks long double into integer and fraction elements.
- (11) The functions `ceil()` and `ceilf()` round up the given float number, whereas the functions `floor()` and `floorf()` round down the float number.
- (12) The function `abs()` returns the absolute value of an integer. The `fabs()` returns the absolute value of a floating-point number and `labs()` returns the absolute value of a long number.

## EXERCISES

### (A) Answer the following questions

- (1) What are the differences between C and C++ functions?
- (2) Describe different parts of function.
- (3) What are void functions?
- (4) What does it mean by function prototype? Is it compulsory?
- (5) When is the function prototype declaration not necessary?
- (6) What are default arguments?
- (7) Where are the default arguments assigned?
- (8) How are the default arguments entered at run time?
- (9) What are inline functions? Discuss its advantages and disadvantages.
- (10) What are the *rvalue* and *lvalue* in an expression? Explain with examples.
- (11) What is the difference between call by value and call by reference? Illustrate them with examples.
- (12) What are constant arguments?

- (13) How is the value of a constant variable changed?
- (14) What is function overloading?
- (15) What are the rules for defining overloaded functions?
- (16) What precautions should we take while overloading function?
- (17) What is the difference between pointer and reference variable?
- (18) What is the difference between normal function and inline function?
- (19) What are actual and formal arguments?
- (20) How can a return statement pass more than a single value from a function?
- (21) What is recursion?
- (22) Explain the rules of recursion?

**(B) Answer the following by selecting the appropriate option**

- (1) The concept of declaring same function name with multiple definitions is
  - (a) **function overloading**
  - (b) operating overloading
  - (c) both (a) and (b)
  - (d) none of the above
- (2) The default arguments are used when
  - (a) **function is called with less arguments**
  - (b) function is void
  - (c) when arguments are passed by reference
  - (d) none of the above
- (3) The constant function
  - (a) **cannot alter values of a variable**
  - (b) can alter values of a constant variable
  - (c) makes its local variable constant
  - (d) none of the above
- (4) The function `abs()` returns
  - (a) **absolute value**
  - (b) negative value
  - (c) both (a) and (b)
  - (d) none of the above
- (5) The use of parentheses is optional for one of the following statement:
  - (a) `return`
  - (b) `main`
  - (c) **`clrscr`**
  - (d) `exit`
- (6) The following program will generate an error message
 

```
void main()
{
 return 0;
}
```

  - (a) **main() cannot return value**
  - (b) void keyword is not allowed in `main()`
  - (c) function should return a non-zero value
  - (d) return statement is not allowed
- (7) Which of the following statements are true?
  - (1) A return type for void specifies that no value be returned
  - (2) Functions by default return `int` value
  - (3) The return type can only be `int`, `char`, or `double`
  - (a) **(1) and (2)**
  - (b) (1), (2), and (3)
  - (c) (1) and (3)
  - (d) none of the above
- (8) C++ provides inline functions to reduce function call overhead, mainly for
  - (a) **small functions**
  - (b) large functions
  - (c) member function
  - (d) none of the above
- (9) To \_\_\_\_ an inline function, we must change it to an outline function.
  - (a) debug
  - (b) edit
  - (c) **remove**
  - (d) comment out
- (10) Identify which of the following function calls are allowed. The prototype declaration is `int sum(int j, int k = 4, int l = 3)`
  - (1) `sum(2)`
  - (2) `sum(2, 3)`
  - (3) `sum()`
  - (4) `sum(3, 4, 5)`
  - (a) **1, 2, and 4**

- (b) 2, 3, and 4
  - (c) 1 and 5
  - (d) 2 and 4
- (11) It is possible to overload functions
- (a) when they have similar name and return type
  - (b) when they have similar name, return type, and different arguments
  - (c) when they have similar name, different argument type, and different return type
  - (d) **when they have similar name, different type of argument, or different number of arguments and return type does not matter**
- (12) What will be the output of the following program?
- ```
int sub(int q)
{
    int m;
    m = sub(q - 1);
    return(m);
}
void main()
{
    int r = sub(7);
    cout<<r;
}
```
- (a) 0
 - (b) **stack overflow**
 - (c) compilation error
 - (d) -5
- (13) Compiler may reject the request for inline function if
- (a) it is small
 - (b) the return type is void
 - (c) **it is a loop, switch, or goto exists**
 - (d) never
- (14) The default return type of main function is
- (a) **int**
 - (b) double
 - (c) long int
 - (d) void
- (15) Advantage of inline function is
- (a) **faster execution**
 - (b) saves memory
 - (c) both (a) and (b)
 - (d) none
- (16) Default arguments are defined when functions are
- (a) defined
 - (b) **declared**
 - (c) both (a) and (b)
 - (d) never

(C) Attempt the following programs

- (1) Write a program to define a function with default argument. Whenever the function needs default values of arguments, it should prompt the user to enter a default value. Also display the default values.
- (2) Write a program to define a constant variable. Change the value of the variable using pointer. Display the default and the newly assigned values along with memory locations.
- (3) Write a program to calculate the power of a given number. Define user-defined function `power()`. Make it inline.
- (4) Write a program to accept a double number. Separate its integer and fractional part.
- (5) Write a program to calculate the absolute value of long and float number.
- (6) Write a program to round down and round up the floating-point number.
- (7) Write a program to overload the function `uabs()`. The function should return absolute value of the given number for data type `int` and `float`.
- (8) Write a program to enter quantity and rate through the keyboard. Then calculate the amount by multiplying quantity and rate. If the fractional part of the amount is greater than or equal to **.50** then round up the number, otherwise round down the number. Enter a minimum of 10 records. Calculate the total fractional parts of the

- amount rounded up and rounded down separately.
- (9) Write a program to display only the integer portion of the given `float` numbers without type casting.
 - (10) Write a program to accept a `float` number through the keyboard. Then calculate the square of the number. Separate the `float` number into integer and fractional part. Individually calculate the square of an integer and fractional parts and add them in another variable. Compare the two squares obtained. Write your observation regarding the squares obtained. (*Note: set precision to 2*).
 - (11) Write a program to calculate the square root of 1 to 10 numbers. Display the sum of integer parts and fractional parts of the square roots obtained. (*Note: set precision to 2*.)
 - (12) Write a program using function overload to convert an integer number to an ASCII character and `float` to ASCII string.
 - (13) Write an inline function to display lines of different patterns.
 - (14) Write a program to display the Fibonacci series up to 21 using recursion.
 - (15) Write a program to display the sum of first 100 odd numbers using recursion.
 - (16) Write a program to find the power of integer number using `pow()` library function.
 - (17) Write a program to return more than one values from function using call by reference method.

This page is intentionally left blank.