Unit 4

Functions in C++

1. Introduction

Functions are the basic building blocks for writing C/C++ programs. Breaking up a program into separate functions, each of which performs a particular task, makes it easier to develop and debug a program.

▶Advantages of using Functions

1. Functions allow breaking down the program into discrete units.
2. Programs that use functions are easier to design, program, debug and maintain.
3. It is possible to separately compile functions.
4. Functions allow for reuse of code.
5. The use of functions allows parallel and fast program development.

You have already studied functions in C. Functions in C++ are declared, defined and called in the same manner. However, there are several differences between functions in C and functions in C++. C++ supports various types of functions as listed below:

1. Call by reference and return by reference
2. Functions with default arguments
3. Function overloading

4. Inline functions
5. Static functions
6. Friend functions
7. Function overriding
8. Virtual and pure virtual functions
9. Template functions

We will study some of these concepts in this chapter and some functions in later chapters.

2. Call By Reference And Return By Reference

There are two methods by which parameters are passed to a function:

1. **Call by value:** The value of the actual argument gets copied into the formal or dummy argument. Any changes made to the dummy argument will not have any effect on the actual parameter.

In the following program, two variables are passed to a function 'swap' by value. The function swaps the two values. From the output, we can see that the variables a and b do not change. This is because the values of a and b are copied into dummy variables x and y. The values of x and y are interchanged but a and b remain the same.

Program : Call by Value

```
#include<iostream>
int main()
{
   int a=10, b=20;
   void swap(int x, int y) ; //function prototype
   swap(a,b);
   cout << "a = " << a << ", b = " << b << endl;
   return 0;
}
void swap(int x, int y) //x and y are local variables
{
      int temp ; //local variable
      temp = x ; x = y ; y = temp;
}
```

Output

a = 10, b =20

2. **Call by reference:** The actual parameter is passed to the function. The formal or dummy argument refers to the actual argument. Any changes made to the dummy argument will also change the value of the actual parameter.

> **Note** In C, all parameters are passed to the function by value. C++ supports both methods of passing parameters to functions.

To implement call by reference, we use reference variables. A reference variable is nothing but an alias of the original variable. If the formal parameters are declared as reference variables, then they will refer to the actual parameter. Any changes made to the formal parameters will be made to the actual variable. The following *example* illustrates call by reference. The swap function interchanges the values of two variables which are passed to the function by reference.

Program : Call by Reference

```
#include<iostream>
int main()
{
   int a=10, b=20;
   void swap(int& , int& ) ; //function prototype
   swap(a,b);
   cout << "a = " << a << ", b = " << b << endl;
   return 0;
}
void swap(int& x, int& y) //x and y are reference variables
{
     int temp ; //local variable
     temp = x ; x = y ; y = temp;
}
```

Output

a = 20, b =10

In this program, x and y are reference variables. They are other names given to variables a and b. Any changes made to x and y are made to a and b respectively. They are initialized as follows:

```
int& x = a;
int& y = b;
```

2.1 Return by Reference

Functions in 'C' can return only one value. Functions in C++ can return a value or a variable. A function can accept reference variables as parameters, it can also return a variable by reference.

Syntax:

```
datatype& function-name(arguments)
{
   //function code
}
```

Consider the following function declaration:

```
int& max(int &x, int &y);
```

Here, max is a function which accepts two reference parameters and returns a reference.

```
int& max(int& x, int& y)
{
     if(x > y)
       return x;
     else
       return y;
}
```

Since the return type of max() is a reference variable, the function returns either x or y whi reference variables. Hence, the function does not return a value but a variable. This means t can use this variable on the left hand side of an assignment statement. *For example*, consic following statement:

```
max(a, b) = -1;
```

This statement assigns –1 to a, if it is larger, otherwise –1 to b.

Note The function cannot return a reference to a local variable i.e. a variable declared within the function. Atleast one argument to the function must be a reference variable.

3. Function Overloading

In C, two functions cannot have the same name in a program. In C++, we can define mu functions having the same name.

Definition:

Function overloading means defining multiple functions having the same name, which perfor same logical task on different parameters.

Function overloading is one way to achieve polymorphism in C++. It is an *example* of compile-time polymorphism.

Note C++ allows both functions and operators to be overloaded.

Overloaded functions must adhere to the following rules:

a. All overloaded functions must have the same name.

b. The argument list of each of the functions must be different.

c. The compiler does not use the return type of the function to distinguish between overloaded functions.

For example, suppose we have to find the product of two numbers which could be either int or double. For this, we have to define four functions, each with the same name but taking parameters of different types.

The following are the function prototypes:

```
int    product(int x, int y);          //function 1
double product(int x, double y);       //function 2
double product(double x, int y);       //function 3
double product(double x, double y); //function 4
```

The appropriate function is selected depending on the values passed in the function call.

```
cout << product(5.2, 10) ;          //calls function 3
cout << product(5.2, 7.1) ;         //calls function 4
cout << product(2, 5) ;             //calls function 1
cout << product(15, 6.32) ;         //calls function 2
```

The selection of a matching overloaded function is a complex process. The compiler follows these steps to correctly select a function:

1. *Exact match*: The compiler first tries to find an exact match based on the arguments.

2. *Match using promotions*: If it does not find an exact match, the compiler uses argument promotions i.e. from char to int, short int to int, float to double.

3. *Match using standard conversions*: Standard conversion involves converting one primitive type into another. It also includes conversion of pointer to derived class into a pointer to base class.

4. *Match using user defined conversions*: User defined conversion functions are usually used for converting from one class type to another either using a constructor of the class or an explicit conversion function.

The following *example* overloads a function for integers and floating point numbers.

Program : Function Overloading

```
#include<iostream.h>
void product(int x, int y)
{
  cout<<"The arguments are <int,int>";
  cout<<"Result is "<< x*y <<endl;
}
void product(int x, double y)
{
   cout<<"The arguments are <int,double>";
   cout<<"Result is "<< x*y <<endl;
}
void product(double x, int y)
{
   cout<<"The arguments are <double,int>";
   cout<<"Result is "<< x*y <<endl;
}
void product(double x, double y)
{
   cout<<"The arguments are <double,double>";
   cout<<"Result is "<< x*y <<endl;
}
int main()
{
   product(10,20);
   product(20.4, 15);
   product(10.0,'A');
   product('A','B');
   return 0;
}
```

Output

```
The arguments are <int,int> Result is 200
The arguments are <double,int> Result is 306
The arguments are <double,int> Result is 650
The arguments are <int,int> Result is 4290
```

Note Type promotions are done in the last two function calls.

A class can also have its member functions overloaded. In the following program, we have defined a class called 'printdata' with three member functions all with the same name 'print'.

i. void print(int) - displays value - <int>, that is, value followed by the value of the integer.

ii. void print(char, int) - displays the character specific number of times.

iii. void print (int, int) – outputs value – [<int>, <int>], that is, value followed by the two integers separated by comma in square brackets.

iv. void print(char *) – outputs value –"char*", that is, value followed by the string in double quotes.

Program : Function Overloading in Class

```
#include<iostream.h>
class printdata
{
public:
  void print(int x)
  {
    cout<<"value-"<<x<<endl;
  }
  void print(int x, int y)
  {
    cout<<"value-[ "<<x<<y<<" ]"<<endl;
  }
  void print(char ch, int n)
  {
    for(int i=1; i<=n;i++)
    cout<<ch;
    cout<<endl;
  }
  void print(char *str)
  {
    cout<<"value-\""<<str <<"\""<<endl;
  }
};
int main()
{
  printdata p;
  p.print(10);
  p.print(10,20);
  p.print('*',5);
  p.print("Hello");
 }
```

Output

```
value-10
value-[ 10 20 ]
*****
value-"Hello"
```

4. Default Arguments

The arguments of a function can be assigned default values in the **function declaration**. The def value will be used if that parameter is not passed any value in the function call. If a value is passe the parameter in the function call, the default value is ignored.

Let us take an *example* to understand this concept. Suppose, we want to find the sum of two numb x and y, the function is:

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

Now, if we want to calculate the sum of three numbers x, y and z, we will have to define anot function with three parameters. Instead, we can modify the above function and pass it th parameters x, y and z. If the user wants to calculate the sum of only x and y, z should be assigne default value of 0. If the user passes three values, then they will be assigned to x, y and respectively. The function will now look like:

```
int sum(int x, int y, int z=0) //default value
{
   return x + y + z;
}
```

The function can be called in two ways:

```
1.   cout << sum(25, 45); //assigns x = 25, y = 45, z = 0
2.   cout << sum(5,45,15); //assigns x = 5, y = 45, z = 15
```

▶ Rules for Default Arguments

1. Only the trailing arguments can have default values. We cannot provide a default value to argument in the middle of an argument list.
2. Default values should be specified only during function declaration (if it is declared a defined separately).
3. The values passed during function call are assigned to formal arguments from left to right.
4. Default argument values should be specified as constants.
5. The constants must be of the same type as the argument.

▶ Advantages of Default Arguments

1. New parameters can be added to existing functions.
2. Functions performing the same task can be combined into a single function using default arguments.

Some *examples* of function declaration with default values are given below:

```
void func1(int x, int y, z=0);          // valid
void func2(int x, int y=0, int z=0);    // valid
void func3(int x=0, int y=0, int z=0);  // valid
void func4(int x = 0, int y)              // invalid
void func5(int x = 0, int y, int z = 0);// invalid
void func6(int x = 0, int y=0, int z);  // invalid
```

Consider the following function declaration:

```
void func(int x=0, int y=0, int z=0); // valid
```

It can be called in four ways:

```
i.   func();  // x=0, y=0, z=0
ii.  func(10);    //x=10, y=0, z=0
iii. func(10,20); // x=10, y=20, z=0
iv.  func(10,20,30); // x=10, y=20, z=30
```

The following program demonstrates the use of default values

Program : Default Values in Functions

```
#include<iostream.h>
void func(int x = 1, int y = 2, int z=3) //default values
{
  cout<<"x = "<<x <<" y = "<<y <<" z = "<<z <<endl;
}
int main()
{
   func();
   func(10);
   func(10,20);
   func(10,20,30);
   return 0;
}
```

Output

```
x = 1  y = 2  z = 3
x = 10 y = 2  z = 3
x = 10 y = 20 z = 3
x = 10 y = 20 z = 30
```

5. Inline Functions

We are familiar with macros in C. A macro is a small piece of code which is replaced where the macro name is used in a program. The macro substitution directive #define defining a macro. Macros can be argumented. Here are a few *examples* of macros.

```
1.  #define SQR(x)      (x)*(x)
2.  #define CUBE(x)     (x)*SQR(x)
3.  #define MAX(x,y)    x>y?x:y
```

There are several advantages as well as disadvantages associated with macros.

▶ Advantages of Macros

1. Since macros are substituted, execution is faster.
2. Macros improve program readability.
3. Very suitable for small, often repeated code.

▶ Disadvantages of Macros

1. Macros are substituted and not compiled.
2. No type checking is done.
3. Macros can lead to side effects which may cause unpredictable or wrong results.
4. The code size increases since macros are substituted.

The alternative to macros is to use functions. Functions are compiled; hence strict type c done. However, there are overheads associated with function calls and return which sl execution.

▶ Advantages of Functions

1. Functions are compiled hence strict type-checking is done.
2. Function code is not replaced, hence program size does not increase.
3. Suitable for large pieces of code also.
4. Don't have side-effects.

▶ Disadvantages of Functions

1. Require additional memory and time for function call and return. Hence execution is

5.1 Inline Functions

The speed advantage of a macro and the compilation, type-checking advantage of a function can be obtained by declaring a function "**inline**". The benefits of macros and functions are combined in inline functions.

Definition

An inline function is a function which works like a macro i.e. its code is replaced at the point where the function is called. An inline function is expanded inline when it is invoked.

The format for an inline function is:

```
inline return-type function-name(arguments ...)
{
   //function code
}
```

To make a function inline, the keyword, "inline" precedes the function prototype and function definition. The call is just like a call to any other function. It is not necessary to use the *inline* keyword during function call.

Note The compiler can ignore the inline keyword and treat the function like a regular function.

Functions containing the following would not be treated inline by the compiler:

1. Static Variables
2. Loops
3. A switch statement
4. Arrays
5. Recursive calls to itself

The following *example* uses an inline function to find the maximum of two numbers.

Program : Inline Function

```
#include<iostream.h>
using namespace std;
inline int max(int x, int y)

  return x>y?x:y;
```

```
int main()
{
    cout << max(20,10) << endl;
    cout<< max(34,45) <<endl;
    return 0;
}
```

5.2 Making Class Functions Inline

All member functions of a class which are defined within the class are treated as inline by default. There is no need to explicitly use the inline keyword. However, if the member function is defined outside, it can be made inline by using the prefix "inline" to the function header.

The following program defines a class point with two inline member functions accept() and display(). The accept function is defined within the class; hence it is treated as inline by default. The display function is defined outside the class and has to be explicitly declared as inline.

Program : Inline Member Functions

```
#include<iostream.h>
class point
{
   int x ;
   int y ;
public:
   void accept()    //function definition  inline by default
   {
      cout<<"Enter the x and y coordinates : "<<endl;
      cin >> x >> y;
   }
   void display();  //function declaration
};
inline void point::display() //defining function outside class
{
     cout <<"x = "<< x <<", y = " << y << endl;
}
int main()
{
     point p ;
     p.accept() ;
     p.display() ;
     return 0;
}
```

Output

```
Enter the x and y
coordinates: 100 200
x = 100, y = 200
```

▶ Advantages of Inline Functions

1. Inline functions are expanded at the point of invocation. Hence, their execution is faster.
2. They are compiled, unlike macros.
3. They have no side-effects.
4. Eliminates function call and return overheads.

▶ Disadvantages of Inline Functions

1. Size of the program increases due to code substitution.
2. Any code that invokes an inline function should be recompiled when the function code changes.
3. The use of the inline keyword is just a request to the compiler; the compiler may not treat the function as inline.
4. The use of inline class member functions violates basic software engineering principles:
   - Separation of interface and implementation.
   - Hiding implementation details.

5.3 Passing and Returning Objects from Functions

We can pass objects of a class to a function as parameters.

Syntax:

```
returntype function-name(classname object)
{
   //function code
}
```

The following example shows how we can pass an object of a class to a function. Consider the class below:

```
class student
{
    int rollno;
    char name[80];
    float perc;
  public:
    //functions
};
```

If we want to write a function to compare the percentages of two student objects s1 ar object will call the function and one object will be passed to the function as a paran *example*,

```
void main()
{
  student s1, s2;
  ....
  if(s1.compare(s2))
    cout<<"Equal percentage";
  else
    cout<<"Not equal";
}
```

The function definition will be as follows:

```
class student
{
  ...
  public:
    int compare(student other)
    {
      if(perc==other.perc)
        return 1;
      else
        return 0;
    }
};
```

If the function has to modify the object, the object must be passed by reference. Cons example where we want to copy one object to another.

```
class student
{
  ...
  public:
    void copy(student& other)
    {
      other.rollno=rollno;
      strcpy(other.name,name);
      other.perc=perc;
    }
};
void main()
{
  student s1, s2;
```

```
    s1.accept();
    s1.copy(s2); //copy s1 to s2
    s2.display();
}
```

Similarly, a function can also return an object. The object can be returned by value or by reference. Consider the fraction class defined below. The 'add' function adds two fraction objects and returns a fraction object. The function 'max' compares two fractions and returns the larger fraction by reference.

```
class fraction
{
    int numerator, denominator;
  public:
    fraction add(fraction f)
    {
      fraction temp;
      temp.numerator = numerator+f.numerator;
      temp.denominator = denominator +f.denominator;
      return temp;
    }
    fraction&  max(fraction& f)
    {
      if(numerator*f.denominator > denominator*f.numerator)
           return *this; //return implicit object
      else
           return f; //return second object
    }
   void modify()
    {
      cin>>numerator>>denominator;
    }
};
void main()
{
   fraction f1, f2, f3;
   ...
   f3=f1.add(f2);
   f1.max(f2).modify();
}
```

Here, the first call to add function adds f1 and f2 and returns the addition. The second call to max function returns the larger fraction and the returned fraction object calls the modify function.

> **Note** Local objects (objects created within the function) cannot be returned by reference.

6. Static Class Members

The data and function members of a class can be declared *static* in C++. Static members do not belong to the object but belong to the class. Hence, they are called **class members**. All the objects of the class share the same copy of static members.

Their features are:

1. Static members are accessed directly by applying the scope resolution operator (::) to the class name.
2. To use static members of the class, there is no need to create class objects.
3. Only one copy of the static member is created and all objects share the same member.

▶ Static Data Members

A data member of a class can be qualified as static. A static data member belongs to the class and hence is also called as a **class variable**. Static variables are normally used to create variables that need to be used by all objects of the class. A static member variable has certain special characteristics. These are:

- Static data members have to be initialized.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class.
- It is visible only within the class, but it exists as long as the program is running.

The syntax of creating and initializing static data members is given below:

```
class classname
{
   ...
  static datatype member;   //static data member
   ...
}

datatype classname :: member = value; //initialization
```

The following program illustrates the use of a static data member. In this program, the student class has one static data member "teacher" which is initialized to "Jane". All objects of the class will share the same teacher variable.

Program : Static Data Members

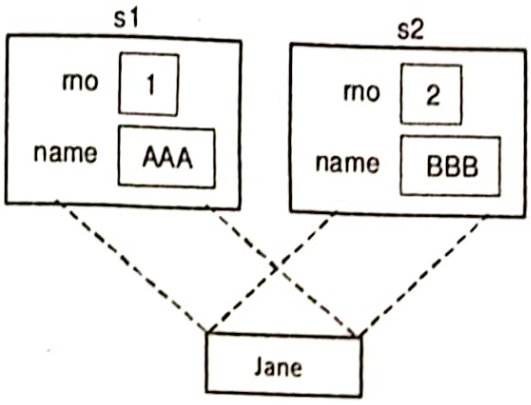
```
#include<iostream.h>
class student

  int rno;
```

```
    char name[20];
    static char teacher[20];  //static data member declaration
    public:
      void accept()
      {
        cout<<"Enter the roll number and name :";
        cin >> rno >> name;
      }
      void display()
      {
        cout<<"Teacher Name = "<< teacher<<endl;
        cout<<"Roll number= "<<rno<<endl;
        cout<<"Name = "<<name;
      }
};
char student::teacher[20]="Jane";//initialization
int main()
{
    student s1, s2;
    s1.accept();
    s2.accept();
    s1.display();
    s2.display();
    return 0;
}
```

Output

```
Enter the roll number and name:
1  AAA
Enter the roll number and name:
2  BBB
Teacher Name = Jane
Roll Number = 1
Name = AAA
Teacher Name = Jane
Roll Number = 2
Name = BBB
```

The following figure shows how a static variable is shared by the two objects.



Sharing of a static data member

n the above program, teacher is a private static data member. If it is declared public, it can be ıccessed directly in main as follows:

```
:out<< " Teacher Name = "  <<  student::teacher;
```

▶ Static Member Functions

Like *static* member variable, we can also have *static* member functions. A static member functio can only access other static members of the class. A static member function cannot access non-stat members of the class since they belong to an object and an object has to be created in order to us them. Whereas, static members don't require an object in order to be used.

A member function that is declared *static* has the following properties.

1. A *static* function can have access to only other static members (functions or variable declared in the same class.
2. A *static* member function can be called using the class name and scope resolution operator.
3. No object needs to be created to invoke a static member function of a class.

The following program illustrates the implementation of these characteristics. The *static* functio **displayteacher()** displays the teacher name which is a static data member of the class.

Program : Static Member Functions

```
#include<iostream.h>
class student
{
  int rno;
  char name[20];
  static char teacher[20];  //static data member
  public:
  void accept()
  {
    cout<<"Enter the roll number and name :"<<endl;
    cin>>rno>>name;
  }
  void display()
  {
    cout<<"Roll number = "<<rno<<endl;
    cout<<"Name = "<<name<<endl;
  }
  static void displayteacher()  //static member function
  {
    cout<<"Teacher Name = "<<teacher<<endl;
  }
};
char student::teacher[20] = "Jane";
```

Output

```
Teacher Name = Jane
Enter the roll number
and name:
1  AAA
Roll number = 1
Name = AAA
```

```
int main()
{
  student::displayteacher();  //invoke static function
  student s1;
  s1.accept();
  s1.display();
  return 0;
}
```

7. Friend Functions and Friend Classes

Private data members of a class can only be accessed by member functions of the class. However, there may be situations in which we may want a non member function to access the private data members.

Definition

A friend function is a non-member function of a class which can access private data members of the class.

For example, consider two classes, *dist1* which stores distance in km and *dist2* which stores distance in miles. We would like to add two objects of these classes. Hence, the add() function will have to access the private data members of both the classes. Hence, it should be declared as a friend of both classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the class.

Syntax:

```
friend return_type function_name(arguments);
```

Note The keyword friend is only used in the function declaration.

For example:

```
class A
{
    . . . .
  public:
    . . . .
      friend void func1(); // declaration
};
```

Here, function *func1* is declared as a friend of the class A. The function declaration is prece keyword *friend*. The function is defined elsewhere in the program like a normal C++ functio function definition does not use either the keyword friend or the scope operator (::). Th friend function is not a member function but it has full rights to access the private members class.

A friend function has the following features:

1. It is not a member function of the class.
2. It has to be declared using the keyword friend.
3. The friend keyword should not be used in the function definition.
4. Since it a non member function, it cannot be invoked by an object of the class.
5. The object of the class must be passed explicitly to the friend function.
6. It cannot access the data members directly and has to use the syntax object.data-member.
7. It can be declared either as public or private.
8. The 'this' pointer cannot be used to access data members of the object since there implicit object.

In the following program, we have a class called Dist1 which has two data members x a Suppose we wish to calculate the distance between two point objects p1 and p2, we can de function called "distance" which needs to access the private data members of both point obje can be declared as a friend function.

Program : Friend Functions

```
#include<iostream.h>
class dist2; //forward declaration of class
class dist1
{
   float km;
   public:
   void accept()
   {
      cout<<"Enter the distance in km:";
      cin>>km;
   }
   friend void add(dist1 d1, dist2 d2); // friend function
};
```

```
class dist2
{
  float  miles;
  public:
  void accept()
  {
    cout<<"Enter the miles :";
    cin>>miles;
  }
  friend void add(dist1 d1, dist2 d2); // friend function
};
void add(dist1 d1, dist2 d2)
{
  dist1 sum1; dist2 sum2;
  sum1.km = d1.km+d2.miles*1.6;
  sum2.miles = d1.km/1.6 + d2.miles;
  cout<<"The sum = "<< sum1.km << " km"<<endl;
  cout<<"The sum = "<< sum2.mi << " miles"<<endl;
}
int main()
{
  dist1 d1; dist2 d2;
  d1.accept();
  d2.accept();
  add(d1,d2);
  return 0;
}
```

Output

Enter the distance in km : 2
Enter the miles : 2
The sum = 5.2 km
The sum = 3.25 miles

In the above program, add() function is not a function of any class. Member functions of one class can be declared as friend functions of another class. *For example*, a function f() of class p can be declared as a friend of class q. In this case, f() can access private members of class q. If a function f() of class p is to be declared as a friend of class q, then the following steps must be followed.

1. Forward declare class q.
2. Define class p and only declare its member functions.
3. Define class q and declare friend functions using p:: function-name.
4. Define member functions of class p.

The following outline illustrates how this can be done.

```
class q; //forward declaration
class p
{
   . . .
   int f();    // member function of p
   . . .
};
class q
{
. . .
   friend int p :: f(); //f() of p is friend of q
. . .
};
int p::f()
{
  //code
}
```

To illustrate this concept, let us consider a class called fraction (numerator and den another class called integer (number). We want to define a member function add() in fr add an integer object to a fraction object. It will need to access the private data member class. Hence, it will be declared as a friend of the integer class.

Program : Friend functions

```
#include<iostream.h>
#include<stdio.h>
class integer ;              //forward declaration
class fraction
{
  int number, den;
  public:
     void accept();
     void add(integer i);
};
class integer
{
   int number;
   public:
   void accept()
   {
     cout<<"Enter the value of the integer :";
     cin>>number;
   }
```

```
 friend void fraction::add(integer i); //friend declaration
;
oid fraction::accept()

 cout<<"Enter the numerator and denominator";
 cin>>numer>>den;

oid fraction::add(integer i)

  cout<<"The addition is "<< number + i.number*den <<"/"<<den;

nt main()

 integer i;
 i.accept();
 fraction f;
 f.accept();
 f.add(i);
 return 0;
}
```

Output

Enter the value of the integer : 2

Enter the numerator and denominator : 5 3

The addition is 11/3

Advantages of Friend Functions

1. It can access private data members of the class.
2. Provides additional functionality which is kept outside the class.
3. Provides functions that need data which is not normally used by the class.
4. Allows sharing private class information by a non member function.
5. An object of the class is not required to invoke the friend function.

Disadvantages of Friend Functions

1. Friend functions increase the chance of namespace collision since they do not belong to the class.
2. They aren't inherited. This means that a friend function of a class is not a friend to the derived class.
3. Friend functions don't bind dynamically i.e. they cannot be declared virtual.

7.1 Friend Class

In some cases, all the member functions of a class A need to be friends of another clas such a case, class A can be declared as a friend of class B. This means that all functions can access private data members of class B.

Syntax:

```
friend class classname;
```

Example:

```
friend class A;
```

To declare class A as a friend of class B, the following syntax should be used:

Syntax:

```
class B; //forward declaration
class A
{
   ...
};
class B
{
  ...
friend class A;
}
```

Let us consider a program with two classes – **list** and **node**. All the member functions class need to access the data members of the node class. Hence, the list class should be dec friend of the node class. The program creates a list of 5 nodes and assigns values 10 to 50.

Program : Friend Class

```
#include<iostream.h>
class node;    //forward declaration
class list
{
  node *first;
  public:
    void create();
    void display();
};
class node
{
  int data;
  node *next;
  public:
    friend class list;  //list is a friend class
```

```
;
oid list::create()

 node *temp, *p;
 first = NULL;
 for(int i=1; i<=5;i++)
 {
   node* p = new node;
   p->data = i*10;
   p->next=NULL;
   if(first==NULL)
      first = temp = p;
   else
   {
     temp->next=p;
     temp=p;
   }
 }

oid list::display()

 cout<<"The list is :";
 for(node *temp=first; temp!=NULL; temp=temp->next)
    cout<<"  "<<temp->data;

nt main()

 list l;
 l.create();
 l.display();
 return 0;
}
```

Output

The list is: 10 20 30 40 50

EXERCISES

A. Fill in the blanks

1. A function that has return type void does not return anything.
2. Using inline functions may reduce execution time, but may increase program size.
3. A static function can be invoked using its class name and function name.
4. The return statement is used to pass a value of an expression back to the calling funct
5. A friend function is a non member function of a class but can access private members.
6. Member functions defined inside a class are treated Inline by default.

B. State True or False

1. When calling a function, if the arguments are passed by reference, the function works with actual variables in the calling program. T
2. A C++ function can return multiple values to the calling function. F
3. A function call of a function that returns a value can be used in an expression like any o variable. T
4. We need not specify any return type for a function that does not return anything. F
5. Using inline functions may reduce execution time, but may increase program size. T
6. Member functions defined within a class become inline by default. T
7. A class can be defined as a friend of another class. T
8. When a class A is defined as a friend of class B, member functions of A become memb functions of B. F
9. Static members of a class can be accessed directly using the class name. T
10. Friend functions can access only the public data members of a class. F
11. A function cannot return by reference. F
12. A function can be declared as private. T
13. A static member function can access only the static data members of the class. T

C. Review Questions

1. What is a function? List the advantages of using functions.
2. What is meant by call by reference and call by value?
3. Compare functions and macros.
4. What is the purpose of return statement?
5. Explain how a static member is defined and declared in C++?
6. What is a static class member?
7. Explain the concept of an inline function with an example.
8. Explain the concept of a friend function and friend class with example.
9. List the features of friend functions.

D. Programming Exercises

1. Define overloaded functions in C++ to calculate the area of circle, triangle, square and rectangle.
2. Write a function in C++ to calculate simple interest for a given amount and period. If the user does not provide rate and number of years, use a default value of 10% and 2 years.
3. Define a class in C++ with two data members n and r. Write a function to calculate nC_r. Define a static member function to calculate factorial of a number.
4. Write a function in C++ which accepts two character variables by reference and converts the case if alphabets. Return the variable which is greater (according to ASCII value). Modify the returned variable to '*'.
5. Implement a class 'printdata' with three member functions all with the same name 'print'

   void print(int) - outputs value - <int> , that is ,value followed by the value of the integer

   void print (int, int) – outputs value – [<int>, <int>] , that is, value followed by the two integers separated by comma in square brackets.

   void print(char *) – outputs value –"char*" , that is , value followed by the string in double quotes.

   Write a main function that uses the above class and its member functions.
6. Implement a class 'maxdata' with two member functions both with the same name 'maximum'

int maximum(int, int) – returns the maximum between the two integer arguments

int maximum (int *) – returns the maximum integer in the array of integers

Write a main function that uses the above class and its member functions.

7. Implement a class 'invertdata' with three member functions all with the same name 'invert'

int invert (int) - returns the inverted integer – invert(5438) will return 8345

char * invert (char *) – returns the reversed string – reverse("comp") will return "pmoc"

void invert(int *) – will reverse the array order – An array [5, 7, 12, 4] will be inverted to [4, 12, 7, 5]

8. Define a class "Integer" which has one data member of the type 'int' and another class "Float" with one data member of the type 'float'. Write member functions in both classes to add an Integer object and a Float object. (Hint: Use friend functions)

Answers

A.

1.	void	2.	inline
3.	static	4.	return
5.	friend	6.	Inline

B.

1.	True	2.	False		
3.	True	4.	False		
5.	True	6.	True		
7.	True	8.	False		
9.	True	10.	False		
11.	False	12.	True	13.	True

VISION