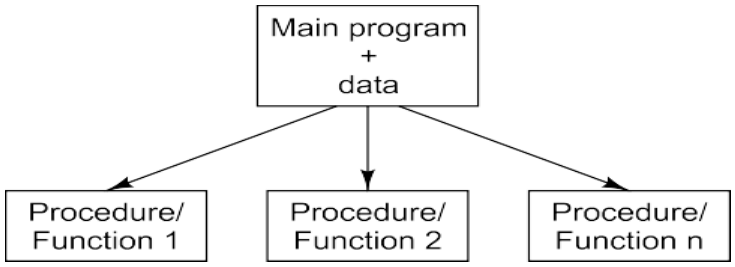
C++ PROGRAMMING

STRUCTURED PROCEDURAL PROGRAMMING SPP

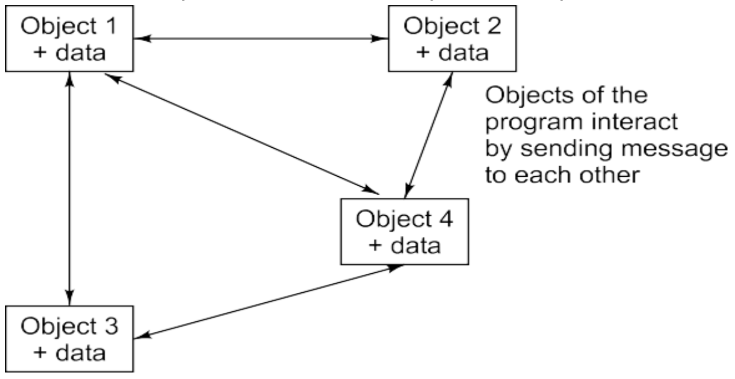
Conventional programming, using high level languages. Such as COBOL, FORTRAN and C is commonly known as structured procedural programming. In the SPP, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on function.



SPP basically consists of writing a list of instruction (or action) for the computer follow and organizing these instructions into groups knows as functions.
SPP basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as functions. We normally use a flowchart to organize these actions and represent the flow of control from one action to another. While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions.
In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data.
Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.
Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really correspond to the elements of the problem.

OBJECT ORIENTED PROGRAMMING OOP

The OOP approach has the advantage of producing better structured and more reliable software's for complex systems, greater reusability, more extensibility, and easy maintainability.


Object 1 + data
Object 2 + data
Objects of the program interact by sending message to each other
Object 4 + data
Object 3 + data

OOP treats data as a critical element in the program development and does not allow it to flow free around the system.

Some of the feature of object-oriented programming are:

1. Programs are divided into what are known as objects.
2. Data structure are designed such that they characterize the objects.

3. Data is hidden and cannot be accessed by external functions.
4. Objects may communicate with each other through functions.
5. New data and functions can be easily added whenever necessary,
6. Follow bottom-up approach in program design.

Basic concepts of object-oriented programming

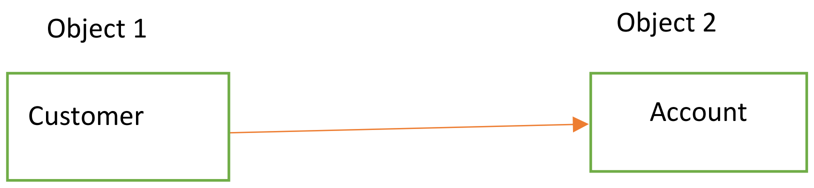
It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

1. Objects
2. Classes
3. Data abstraction and encapsulation
4. Inheritance
5. Polymorphism

Objects

Objects a are the basic run-time entities in an object -oriented system. They may represent a person, a place, a bank account etc.

Example



When a program is executed, the object interacts by sending message to one another. For example, if "Customer" and "Account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data and code to manipulate the data.

Although different authors represent them differently, below fig two notations that are popularly used in object-oriented analysis and design.

Object: student
Data Name Date of birth Marks --------- ---------
Function void Total() void Average() void Display() ---------- ----------

Classes

The entire set of data and code of an object can be made a user-defined data type with the help of a class.
Class is the user defined data type.
Object are the variable of class type.
Once a class has been defined, we can create any number of objects belonging to that class.
A class is a collection of objects of similar types.
Example, Mango, apple and orange are objects of the fruit class.

Encapsulation

The wrapping up of data and function into a single unit (called class) is know as encapsulation. The data is not accessible outside world, only these functions can access data which are wrapped in the class.

Abstraction

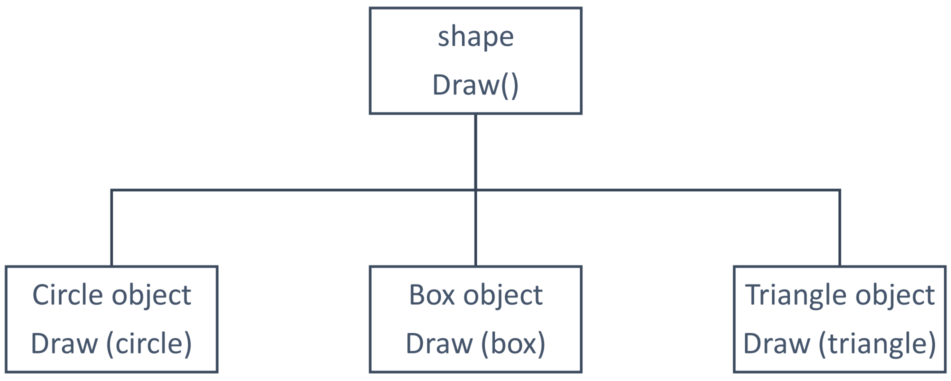
Abstraction refers to the act of representing essential features without including the background details or explanations.
It shows summary or essential info.

Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class.
In OOPs, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it.

Polymorphism

Polymorphism means the ability to take more than one form.
For example,
Consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.
In below example that a single function name can be used to handle different number and different type of arguments, using a single function to perform different types of tasks in known as function overloading.



Introduction

A class is a user-defined data type which holds both the data and functions. The internal data of a class is called member data (or data member) and the functions are called member functions. The member functions mostly manipulate the internal data of a class. The member data of a class should not normally be addressed outside a member function. The variables of a class are called objects or instances of a class.

The word 'class' is a fundamental and powerful keyword in C++. It is significantly useful as it is used to combine the data and operations of a structure into a single entity. The class construct differs from the conventional C language struct construct. The class construct provides support for data hiding, abstraction, encapsulation, single inheritance, multiple inheritance, polymorphism and public interface functions (methods) for passing message between objects. This section stresses only the elementary concepts of the object-oriented programming (OOP) terminology and the subsequent sections deals with implementation of these topics in C++.

(a) ***Data Abstraction:*** In OOP, the data abstraction is defined as a collection of data and methods (functions).

(b) ***Data Hiding****:* In C++, the class construct allows to declare data and methods, as a public, private and protected group. The implementation details of a class can be hidden. This is done by the data hiding principle.

(c) ***Data Encapsulation:*** The internal data (the member data) of a class are first separated from the outside world (the defined class). They are then put along with the member functions in a capsule. In other words, encapsulation groups all the pieces of an object into one neat package. It avoids undesired side effects of the member data when it is defined out of the class and also protects the intentional misuse of important data. Classes efficiently manage the complexity of large programs through encapsulation.

(d) ***Inheritance:*** C++ allows a programmer to build hierarchy of classes. The derivation of classes is used for building hierarchy. The basic features of classes (parent classes or basic classes) can be passed onto the derived classes (child classes). In practice, the inheritance principle reduces the amount of writing; as the derived classes do not have to be written again.

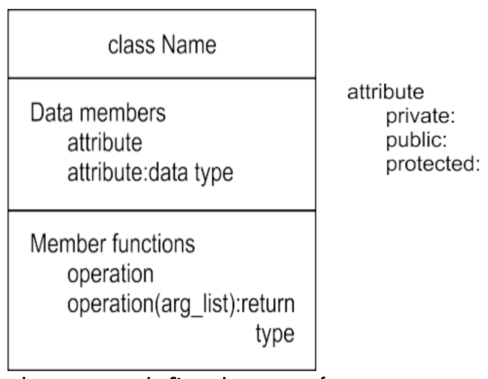
(e) ***Polymorphism****:* In OOP, polymorphism is defined as how to carry out different processing steps by a function having the same messages.

The following equivalent terminology is used between the function-oriented programming and OOP:

Function oriented programming	***Object oriented programming (OOP)***
User-defined types	Classes
Variables	Objects
Structure members	Instance variables
Functions	Methods
Function call	Message passing

STRUCTURES AND CLASSES

A structure contains one or more data items (called members) which are grouped together as a single unit. On the other hand, a class is similar to a structure data type but it consists of not only data elements but also functions which are operated on the data elements. Secondly, in a structure, all elements are public by default, while in a class it is private. The data and functions can be defined in a class as one of the sections such as private, public and protected. Function defined within a class have a special relationship to the member data and member functions (methods).



```
class user_defined_name {
        private :
                data_type members
                implementation operations
                list of friend functions
                list of friend functions
        public :
                data_type members
                 implementation operations
        protected :
                data_type operations
                 implementation operations
};
class user_defined_name variable1, variable2, …variable n;
```

The keyword typedef is not required since a class name is a type of name. The keywords private, protected and public are used to specify the three levels of access protection for hiding data and function members internal to the class.

(a) ***Private****:* In private section, a member data can only be accessed by the member function and friends of this class. The member functions and friends of this class can always read or write private data members. The private data member is not accessible to the outside world (out of the class).

(b) ***Protected****:* The members which are declared in the protected section, can only be accessed by the member functions and friends of this class. Also, these functions can be accessed by the member functions and friends derived from this class. It is not accessible to the outside world.

(c) ***Public:*** The members which are declared in the public section, can be accessed by any function in the outside world (out of the class). The public implementation operations are also called as member functions or methods, or interfaces to out of the class. Any function can send messages to an object of this class through these interface functions.

The public data members can always read and write outside this class. A member function can be inline, which means the member function can be defined within the body of the class constant.

A class may be defined using one of the following keywords:

class
struct
union

A structure is a class declared with the class key struct; its members and base classes are public by default. A union is a class declared with the class key union; its members are public by default and it holds only one member at a time.

For example, *the following declaration illustrates the default member access specifier:*

(1) A class is declared with the keyword 'class',

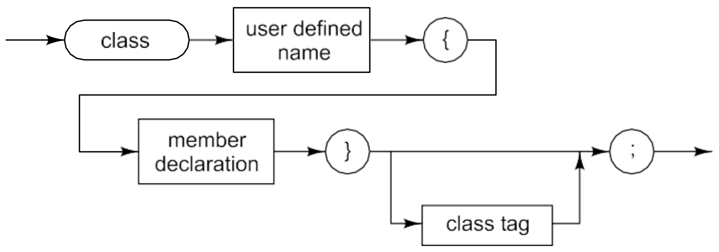
```
class sample {
        int a;
        float x;
        char ch;
        };          // A class by default has all its members private
```
(2) A class is declared with the keyword ‘struct’,
```
        struct sample {
                int a;
                float x;
                char ch;
        };          // A struct by default has all its members public
```
(3) A class is declared with the keyword ‘union’,
```
                union sample {
                        int a;
                        float x;
                        char ch;
                };
```
A union by default has all its members public and it holds only one member at a time.

Class is a key concept of C++. A class is a user-defined type and it is the unit of data hiding and encapsulation. Polymorphism is supported through classes with virtual functions. The class provides a unit of modularity.

DECLARATION OF A CLASS

A class is a user-defined data type which consists of two sections, a private and a protected section that holds data and a public section that holds the interface operations.

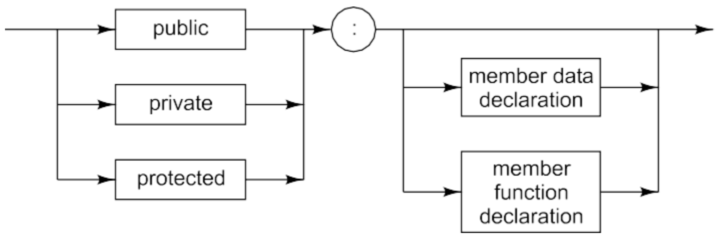
A class definition is a process of naming a class and data variables, and methods or interface operations of the class.

The general syntax of the class construct is:
```
    class user_defined_name {
        private:
                data_type members
                implementation operations
                list of friend functions
                list of friend functions
        public:
                list of friend functions
                data_type members
                implementation operations
        protected:
                list of friend functions
                data_type operations
                implementation operations
    };
    class user_defined_name variable1, variable2, …variable n;
```


A class declaration introduces the class name into the scope where it is declared and hides any class, object, function or other declaration of that name in an enclosing scope.
```
class item {
    // member lists
```

};
Class members can be one of the following member lists:

- data
- functions
- classes
- enumerations
- bitfields
- friends
- data type names



For example,

(1) A class date is defined as day, month and year of a member data variable without any method or any member function.

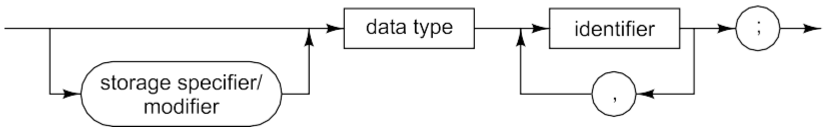
```
class date {
    private:
            int day;
            int month;
            int year;
    };
    class date today;         // today is object of class date
```

(2) *The student's particulars such as rollno, age, gender, height and weight can be grouped as:*

```
class student {
    private:
            long int rollno;
            int age;
            char gender;
            float height;
            float weight;
    public:
            float weight;
            void getinfo ();  //member function
            void disinfo ();   // member function
    };         // end of class definition
```

The class definition is permitted to be declared within the class declaration itself.

```
    class date {
            public:
                    int day;
                    int month;
                    int year;
    } today;        // now the object is created of class date
```



The following declaration is identical while accessing the member of the class.

(1)
```
class sample {
        int x;
        int y;
};      // by default, members are private
```
(2)
```
class sample {
        private:
                int x;
                int y;
};
```
Note that the keyword 'private' is used for declaring the data items of a class explicitly as a private group.

Note, that a member may not be declared twice in the member list. The member list defines the full set of members of the class and no member can be added elsewhere.

The following class declarations are invalid.

(1)
```
class sample {
        private:
                int one;
                int two;
                int one;
};
// error, the data member 'one' has been redeclared
```
(2)
```
class item {
        private:
        float x;
        char ch;
};
item :: y;
// error, no member can be added elsewhere, other than class declaration
```
(3)
```
class sample {
        private:
                int x,y;
        public:
                int one;
                void setdata();
                void getdata();
                void display();
                void setdata();    // error, redeclaration
};
```
The same rule applies for function declaration also.

(4) Note that a single name can denote several function members provided their types are sufficiently different. The same name cannot denote both a member function and a member data.
```
class xy {
        private :
                int funct;
        public :
                int funct();        // error, same name is used for both
};      // data member and function
```
(5)
```
class abc {
        public:
                int funct();
        int (*funct) ();
};
        // error, pointer to a function and the function name are same
```

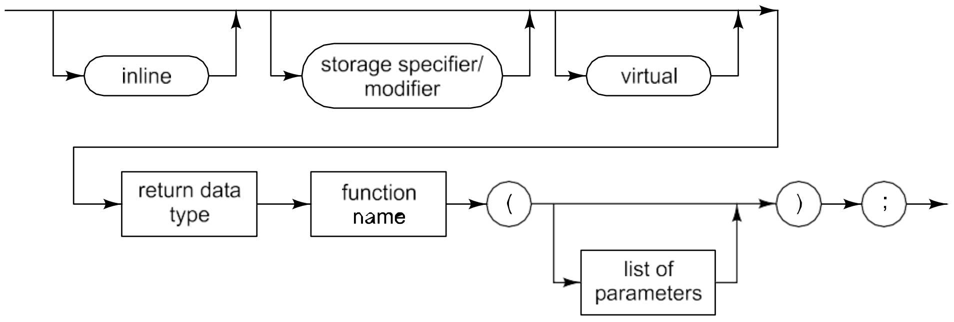
MEMBER FUNCTIONS

A function declared as a member (without the friend specifier) of a class is called as a member function. Member functions are mostly given the attributes of public because they have to be called outside the class either in a program or in a function.

For example, *the following program segments illustrate how a data member and member function are defined in C++:*

```
class sample {
        private:
                int x;
                int y;
        public:
                int sum() {
                        return(x+y);
                }
                int diff() {
                        return(x-y);
                }
};
```

Defining a member function of a class outside its scope: In C++, it is permitted to declare the member functions either inside the class declaration or outside the class declaration. A member function of a class is defined using the :: (double colon symbol) scoping operator.



The general syntax of the member function of a class outside its scope is:
return_type class_name :: member_functions(argument 1,2...n)

For example, the following program segment shows how a member function is declared outside the class declaration. The member functions are defined separately as part of the program.

```
class sample {

        private:
                int x;
                int y;
        public:
                int sum();          // member function declaration
                int diff();          // member function declaration
};          // end of class definition
int sample :: sum()          // member function definition
{
        return(x+y);
}
int sample :: diff()          // member function de nition
{
        return(x-y);
}
```

The use of the scope operator double colon (::) is important for defining the member functions outside the class declaration.

DEFINING THE OBJECT OF A CLASS

In general, a class is a user-defined data type, while an object is an instance of a class template. A class provides a template, which defines the member functions and variables that are required for objects of the class type. A class must be defined prior to the class declaration

The general syntax for defining the object of a class is:

```
class user_defined_name {
        private:
                // methods
        public:
                // methods
        protected:
                // methods
};
user_defined_name object 1,object 2 ... object n;
```

where object 1, object 2 and object n are the identical class of the user_defined_name.

For example, the following program segments show how to declare and to create a class of objects.

(1) Student's information such as roll no, age, gender, height and weight are grouped as

```
class student_info{
        private:
                long int rollno;
                int age;
                char gender;
                float height;
                float weight;
        public :
                void getinfo ();
                void disinfo ();
                void process();
                void personal();
}; // end of class definition student_info obj;
// obj is the object of the class student_info
```

ACCESSING A MEMBER OF CLASS

There are two ways one can access a member of a class similar to accessing member of a struct or union construct. A data or function member of a class construct is accessed using the . (period) operator.

The general syntax for accessing a member of class is

class_object.data_member class_object.function_member

For example,

```
class sample {
        private :
                int x;
                int y;
        public :
                int sum();
                 int diff();
}; // end of class definition
void main (void)
{
        sample one;
        one.sum();        // accessing the member function sum()
        one.diff();       // accessing the member function diff()
        -------
        -------
}
```

PROGRAM 1

```
#include <iostream>
using namespace std;
int main()
{
        class date {
                public :
                        int day, month, year;
        };
        class date today;
        today.day = 10;
        today.month = 5;
        today.year = 2007;
        cout << "Today's date is =" << today.day << "/";
        cout << today.month << "/" << today.year << endl;
        return 0;
}
```

Output

```
Today's date is = 10/5/2007
```

While the keyword public is not used to define the members of a class, the C++ compiler assumes, by default, that all its members are private. The data members are not accessible outside the class. For example, the following program demonstrates the accessibility of the members.

```
// class 2.cpp
#include <iostream>
using namespace std;
int main()
{
        class date {
        // by default, members are private
        int day, month, year;
};
        class date today;
        today.day = 10;
        today.month = 5;
        today.year = 2007;
        cout << " Today's date is = " << today.day << "/";
        cout << today.month << "/" << today.year << endl;
}
```

The following error message will be displayed during the compilation time.

```
date.day is a private
date.month is a private
date.year is a private
```

PROGRAM 2

```
//class with data and member function
#include <iostream>
```

```
using namespace std;
class date {
	private:
		int day,month,year;  public :
		void getdata( int d,int m,int y)
		{
			day = d;  month = m;  year = y;
		}
		void display (void)
		{
			cout << “ Today’s date is = ” << day << “/”;  cout << month << “/” << year << endl;
		}
	};		// end of class definition
int main()
{
	date today;
	int d1, m1, y1;
	d1 = 10;
	m1 = 5;
	y1 = 2007;
	today.getdata(d1,m1,y1);
	today.display();
	return 0;
}
```

Output

```
Today’s date is = 10/5/2007
```

PROGRAM 3

A program to read the data variables of a class by the member function and display the contents of the class objects on the screen.

```
#include <iostream>
using namespace std;
class date {
	private :
		int day,month,year;
	public :
		void getdata()
		{
			cout << “ enter the date ( dd-mm-year) ” << endl;  cin >> day >> month >> year;
		}
		void display ()
		{
			cout << “ Today’s date is = ” << day << “/”;  cout << month << “/” << year << endl;
		}
	};		// end of class de nition
int main()
{
	date today;
	today.getdata();
	today.display();
	return 0;
}
```

Output

```
enter the date (dd-mm-year)
```

```
12 5 2007
Today's date is = 12/5/2007
```

PROGRAM 4

A program to illustrate the use of the simple arithmetic operations such as addition, subtraction, multiplication and division using a member function. These methods are defined within the scope of a class definition.

```
#include <iostream>
using namespace std;
class sample {
        private :
                int x,y;
        public :
                void getinfo(){
                        cout << " enter any two numbers ? " << endl;  cin >> x >> y ;
                }
                void display(){
                cout << " x = " << x << endl;
                cout << " y = " << y << endl;
                cout << " sum = " << sum() << endl;
                cout << " dif = " << diff() << endl;
                cout << " mul = " << mult() << endl;
                cout << " div = " << div() << endl;
        }
        int sum(){
                return(x+y);
        }
        int diff(){
                return(x-y);
        }
        int mult(){
                return(x*y);
        }
        float div(){
                return( ( float)x/( float)y);
        }
}; // end of class definition
int main()
{
        sample obj1;
        obj1.getinfo();
        obj1.display();
        obj1.sum();
        obj1.diff();
        obj1.mult();
        obj1.div();
        return 0;
}
```

Output

```
enter any two numbers?
1       2
x = 1
y = 2
sum = 3
diff = -1
```

```
mul = 2
div = 0.5
```

PROGRAM 5

A program to illustrate the use of the simple arithmetic operations such as addition, subtraction, multiplication and division using a member function. These are defined out of the scope of a class definition.

```
#include <iostream>
using namespace std;
class sample {
        private:
                int x;
                int y;
        public:
                void getinfo();
                void display();
                int sum();
                int diff();
                int mult();
                float div();
                };          // end of class definition
        void sample :: getinfo()
        {
                cout << “enter any two number ? ” << endl;
                cin >> x >> y ;
        }
        void sample :: display()
        {
                cout << “ x = ” << x << endl;
                cout << “ y = ” << y << endl;
                cout << “ sum = ” << sum() << endl;
                cout << “ dif = ” << diff() << endl;
                cout << “ mul = ” << mult() << endl;
                cout << “ div = ” << div() << endl;
        }
        int sample :: sum()
        {
                return(x+y);
        }
        int sample :: diff()
        {
                return(x-y);
        }
        int sample :: mult()
        {
                return(x*y);
        }
        float sample :: div()
        {
                return(( float)x/ ( float)y);
        }
int main()
{
        sample obj1;
        obj1.getinfo();
        obj1.display();
        obj1.sum();
        obj1.diff();
        obj1.mult();
        obj1.div();
        return 0;
}
```

Output

```
enter any two number?
1 3
x = 1
y = 3
sum = 4
dif = -2
mul = 3
div = 0.333333
```

PROGRAM 6

A program to find the area of a circle whose radius is given as input using an OOP technique.

```
#include <iostream>
#include <cmath>
using namespace std;
const float pi = 3.14159;
class circle {
        private :
                float radius,area;
        public :
                void get_radius();
                void find_area();
                void display_area();
};        // end of class definition
void circle ::get_radius()
{
        cout << “enter radius of a circle \n”;
        cin >> radius;
}
void circle ::find_area()
{
        area = pi * radius * radius;
}
void circle :: display_area()
{
        cout << endl;
        cout << “ radius = ” << radius << ‘\n’;
        cout << “ Area of a circle = ” << area << ‘\n’;
}
int main()
{
        circle obj;
        obj.get_radius();
        obj.find_area();
        obj.display_area();
        return 0;
}
```

Output

```
enter radius of a circle
10
radius   = 10
Area of a circle = 314.159
```

PROGRAM 7

A program to find the factorial of a given number using an OOP technique.

```
#include <iostream>
using namespace std;
class abc {
        private:
                int n;
        public:
                long int fact (int n);
};
long int abc :: fact(int n)
{
        long int temp = 1;
        for (int i = 1; i <= n; ++i)
                temp *= i;
        return (temp);
}
int main()
{
        abc obj;
        int max;
        cout << " enter a number \n";
        cin >> max;
        long int total = obj.fact(max);
        cout << "Factorial " << max << "! = " << total;
        cout << endl;
        return 0;
}
```

Output

```
enter a number   5
Factorial 5! = 120
```

PROGRAM 8

A program to read a set of characters from the keyboard and store it in the character array and display its contents onto the video screen using an OOP technique.

```
#include <iostream>
using namespace std;
class abc {
        public:
                char a[200];
                void getdata();
                void display();
        };
void abc :: getdata()
{
        char ch;  int i = 0;
        cout <<"enter a line of text and terminate with @\n";
        while ((ch = cin.get()) != ‘@’) {
                a[i++] = ch;
        }
        a[i++] = ‘\0’;
}
void abc :: display()
{
        cout << “contents of a character array \n”;
        for (int i = 0; a[i] != ‘\0’; ++i)
        cout.put(a[i]);
}
int main()
{
        abc obj;
        obj.getdata();
        obj.display();
        return 0;
}
```

Output

```
enter a line of text and terminate with @
 this is
a test program
by Ravi
@
contents of a character array
this is
a test program
by Ravi
```

ARRAY OF CLASS OBJECTS

An array is a user-defined data type whose members are homogeneous and stored in contiguous memory locations. For practical applications such as designing a large size of data base, arrays are very essential. The declaration of an array of class objects is very similar to the declaration of the array of structures in C++.

The general syntax of the array of class objects is:

```
class user_defined_name {
        private:
                // methods
        public:
                // methods
        protected:
                // methods
};
class user_defined_name object [MAX];
```

Note, MAX refer to number of objects

PROGRAM 9

A program to read students’ particulars such as roll number, age, gender, height and weight from the keyboard and display the contents of the class on the screen. The class ‘student_info’ is defined as an array of class objects. This program shows how to create an array of class objects and how to access these data member and member functions in C++.

```
#include <iostream>
using namespace std;
const int MAX = 100;
```

```
class student_info
{
        private:
                long int rollno;
                int age;
                char gender;
                float height, weight;
        public :
                void getinfo();
                void disinfo();
};              // end of class definition
void student_info :: getinfo()
{
        cout << “Roll no :”;
        cin>> rollno;
        cout <<“Age:”;
        cin >> age;
        cout << “Gender :”;
        cin >> gender;
        cout << “Height:”;
        cin >> height;
        cout << “Weight:”;
        cin >> weight;
}
void student_info :: disinfo ()
{
        cout << endl;
        cout << “Roll no=” << rollno << endl;
        cout << “Age=” << age << endl;
        cout << “Gender=” << gender << endl;
        cout << “Height=” << height << endl;
        cout << “Weight=” << weight << endl;
}
int main()
{
        student_info object[MAX]; // array of objects
        int i, n;
        cout << “How many students? \n” << endl;
        cin >> n;
        cout << “enter the following imformation \n” << endl;
        for (i = 0; i <= n-1; ++i) {
                int j = i;
                cout << endl;
                cout << “record =” << j+1 << endl;
                object[i].getinfo();
        }
        cout << “contents of class \n”;
        for (i = 0; i <= n-1; ++i) {
                object[i].disinfo();
        }
}
```

Output

```
How many students?
2
```

```
enter the following information
record = 1
Roll no: 20071
Age: 21
Gender: M
Height: 170
Weight: 56

record = 2
Roll no: 20072
Age: 20
Gender: F
Height: 160
Weight  : 50

contents of class

Roll no  = 20071
Age = 21
Gender = M
Height   = 170
Weight = 56

Roll no  = 20072
Age = 20
Gender = F
Height = 160
Weight = 50
```

POINTERS AND CLASSES

Pointer is a variable which holds the memory address of another variable of any basic data types such as int, float or sometimes an array. The pointer variable is very much used to construct complex data bases using the data structures such as linked lists, double linked lists and binary trees.

The following declaration of creating an object is valid in C++.

```
class sample {
        private:
                int x;
                float y;
                char s;
        public:
                void getdata();
                void display();
};
sample *ptr;
```

Case 1 A member of class object can be accessed by the indirection operator which has been shown in the following program segment:

```
class student{
        private:
                -------
                -------
        public:
                -------
                -------
    };          // end of class definition
    void main(void)
    {
                student *ptr;
                ------
                ------
                (*ptr).data_member;
                (*ptr).member_function();
    }
```

Case 2 A member of class object can be accessed by the pointer of a class operator which has been shown in the following program segment:

```
    class student{
                private:
                            -------
                            -------
                public:
                            --------
                            -------
    };          // end of class definition
    int main()
    {
                student *ptr;
                --------
                -------
                ptr->data_member;
                ptr->member_function();
                return 0;
    }
```

PROGRAM 10

A program to assign some values to the member of class objects using a pointer structure operator (->).

```
#include <iostream>
using namespace std;
class student_info{
    private:
                long int rollno;
                int age;
                char gender;
                float height;
                float weight;
    public:
                void getinfo ();
                void disinfo ();
    };          // end of class definition
void student_info :: getinfo()
{
    cout << " Roll no :";
    cin >> rollno;
    cout <<" Age:";
    cin >> age;
    cout << "Gender : ";
    cin >> gender;
```

```
            cout << “Height:”;
            cin >> height;
            cout << “Weight:”;
            cin >> weight;
}
void student_info :: disinfo()
{
            cout << endl;
            cout << “ Roll no= ” << rollno << endl;
            cout << “ Age= ” << age << endl;
            cout << “ Gender= ” << gender << endl;
            cout << “ Height= ” << height << endl;
            cout << “ Weight= ” << weight << endl;
}
int main()
{
       student_info a;
            student_info *ptr = &a; // ptr is an object of class
            student cout << “enter the following information” << endl;
            ptr->getinfo();
            cout << “ \n contents of class ” << endl;
            ptr->disinfo();
            return 0;
}
```

Output

```
enter the following information
Roll no : 200710
Age: 23
Gender: M
Height: 176
Weight: 67

contents of class
Roll no=200710
Age = 23
Gender = M
Height=176
Weight =67
```

CLASSES WITHIN CLASSES NESTED CLASS

C++ permits declaration of a class within another class. A class declared as a member of another class is called as a nested class or a class within another class. The name of a nested class is local to the enclosing class. The nested class is in the scope of its enclosing class.

The general syntax of the nested class declaration is shown below.

```
class outer_class_name {
     private :
        // data
      protected :
         //data
         // methods
      public :
         // methods
      class inner_class_name {
           private :
                // data of inner class
            public :
                // methods of inner class
        }; // end of inner class declaration
}; // end of outer class declaration
outer_class_name object1;
outer_class_name :: inner_class_name object2;
```

PROGRAM 11

```
#include <iostream>
#include <string>
using namespace std;
class student_info {
    private :
        char name[20];
        long int rollno;
        char gender;
    public :
        student_info(char *na, long int rn, char gn);
        void display();
        class date {
            private :
                int day,month,year;
            public :
                date (int dy,int mh, int yr);
                void show_date();
             }; // end of date class declaration
       };// end of student_info class declaration
student_info :: student_info(char *na,long int rn,char gn)
{
      strcpy (name,na);
      rollno = rn;
      gender = gn;
}
student_info::date :: date(int dy, int mh, int yr)
{
     day = dy;
     month = mh;
     year = yr;
}
void student_info:: display()
{
     cout << " student's_name Rollno Gender date_of_birth(dd-mm-yr) \n";
     cout << " -------------------------------------------------- " << endl;
     cout << name << " " ;
     cout << rollno << " ";
     cout << gender << " ";
}
void student_info::date::show_date()
{
       cout << day << '/' << month << '/' << year << endl;
       cout << " ------------------------------" << endl;
}
int main()
```

```
{
    student_info obj1("ABC",200710,'M');
    student_info::date obj2(13,7,94);
    obj1.display();
    obj2.show_date();
    return 0;
}
```

Output

```
student's_name   Rollno    Gender    date_of_birth(dd-mm-yr)
---------------------------------------------
ABC                    200710   M      13/7/94
---------------------------------------------
```

Constructor

A constructor is a special member function for automatic initialisation of an object. Whenever an object is created, the special member function, i.e., is the constructor will be executed automatically. A constructor function is different from all other nonstatic member functions in a class because it is used to initialise the variables of whatever instance being created. Note that a constructor function can be overloaded to accommodate many different forms of initialisation.

Syntax rules for writing constructor functions The following rules are used for writing a constructor function:

∑ A constructor name must be the same as that of its class name.
∑ It is declared with no return type (not even void).
∑ It cannot be declared `const` or `volatile` but a constructor can be invoked for a `const` and `volatile` objects.

∑ It may not be `static`.
∑ It may not be `virtual`.
∑ It should have public or protected access within the class and only in rare circumstances it should be declared private.

The general syntax of the constructor function in C++ is,

```
class class_name {
        private:
         ----------
         ----------
    protected :
         ----------
         ----------
        public :
                class_name();    // constructor
};
class_name :: class_name()
{
}
```

The following examples illustrate the constructor declaration in a class definition.

(1)

```
    class employee {
         private :
                        char name[20];
                        int ecode;
                      char address[20];
       public :
                        employee();      // constructor
                        void getdata();
            void display ();
      };
        employee() :: employee();
        {
                ----------
                ----------
    }
```

(2)

```
class account {
        private :
                float balance;
                float rate;
```

```
          public :
                    account()       //constructor
                    {
                         ---------
                         ---------
                    }
                    void create_acct();
};
```

PROGRAM 1

A program to demonstrate how to use a special member function, namely, constructor in C++.

```
#include <iostream>
using namespace std;
class abc
{
     public:
       abc() {
          cout << “for class constructor\n”;
       }
};
int main()
{
    abc obj;
    return 0;
}
```

Output

```
for class constructor
```

PROGRAM 2

A program to generate a series of Fibonacci numbers using the constructor where the constructor member function has been defined in the scope of class definition itself.

```
#include <iostream>
using namespace std;
class fibonacci {
   private :
       unsigned long int f0,f1,fib;
   public :
       fibonacci ()          // constructor
       {
                   f0 = 0;
                   f1 = 1;
                   cout << “Fibonacci series of first 10 numbers\n”;
                   cout << f0 << ‘\t’ << f1 <<‘\t’;
                   fib = f0+f1;
            }
         void increment ()
         {
                   f0 = f1;
                   f1 = fib;
              fib = f0+f1;
            }
            void display()
            {
                    cout <<fib << '\t';
            }
};
// end of class construction
 int main()
{
      fibonacci number;
      for (int i = 3; i <= 10; ++i) {
            number.display();
            number.increment();
      }
            return 0;
}
```

Output

```
Fibonacci series of first 10 numbers
0           1           1           2           3           5           8           13          21          34
```

PROGRAM 3

A program to simulate a simple banking system in which the initial balance and the rate of interest are read from the keyboard and these values are initialised using the constructor member function. The program consists of the following methods:

- *To initialise the balance amount and the rate of interest using the constructor member function*
- *To make a deposit*
- *To withdraw an amount from the balance*
- *To find the compound interest based on the rate of interest*
- *To know the balance amount*
- *To display the menu options*

```
#include <iostream>
using namespace std;
class account {
      private :
          float balance ;
          float rate;
      public :
                  account();          // constructor
                  void deposit ();
                  void withdraw ();
                  void compound();
                  void getbalance();
                  void menu();
}; //end of class definition
account :: account() // constructor
{
            cout << " enter the initial balance \n";
            cin >> balance;
            cout << " interest rate in decimal\n";
            cin >> rate;
}
//deposit
void account :: deposit ()
{
            float amount;
            cout << “ enter the amount : ”;
            cin >> amount;
            balance = balance+amount;
}
//attempt to withdraw
void account :: withdraw ()
{
           float amount;
            cout << “ how much to withdraw ? \n”;
            cin >> amount;
            if (amount <= balance) {
                     balance = balance-amount;
                     cout << “ amount drawn = ” << amount << endl;
                     cout << “ current balance = ” << balance << endl;
           }
            else
                  cout << 0;
}
void account :: compound ()
{
            float interest;
            interest = balance*rate;
            balance = balance+interest;
            cout << “interest amount = ” << interest <<endl;
            cout << “ toal amount = ” << balance << endl;
}
void account :: getbalance()
{
            cout << “ Current balance = ” ;
            cout << balance << endl;
}
void account :: menu()
{
            cout << “ d -> deposit\n”;
            cout << “ w -> withdraw \n”;
            cout << “ c -> compound interest\n”;
            cout << “ g -> get balance \n”;
            cout << “ q -> quit\n”;  cout << “ m -> menu\n”;
            cout << “ option, please ? \n”;
}
int main()
{
            class account acct;
            char ch;
            acct.menu();
            while ( (ch = cin.get()) != ‘q’){
                  switch (ch) {
                  case ‘d’ :
                            acct.deposit();
                            break;
                            case ‘w’ :
                            acct.withdraw();
                            break;
                            case ‘c’ :
```

```
                        acct.compound();
                        break;
                        case ‘g’ :
                        acct.getbalance();
                        break;
                        case ‘m’:
                        acct.menu();
                        break;
                } // end of switch statement
        }
        return 0;
}
```

Output

```
enter the initial balance  1000
interest rate in decimal  0.2
d -> deposit
w -> withdraw
c -> compound interest
g -> get balance
q -> quit
m -> menu option, please?
g
Current balance = 1000
d
enter the amount : 1000
 g
Current balance = 2000
w
how much to withdraw?
500
amount drawn = 500
current balance = 1500
 c
interest amount = 300
 toal amount = 1800
g
Current balance = 1800
q
```

Default Constructors

The default constructor is a special member function which is invoked by the C++ compiler without any argument for initialising the objects of a class. In other words, a default constructor function initialises the data members with no arguments. It can be explicitly written in a program. In case, a default constructor is not defined in a program, the C++ compiler automatically generates it in a program. The purpose of the default constructor is to construct a default object of the class type.

The general syntax of the default constructor function is,

```
class class_name {
        private :
                --------
                --------
        protected :
                --------
                --------
        public :
                class_name();       // default constructor
};
class_name :: class_name () {}         // without any arguments
```

The typical form of the default constructor is:

```
class_name :: class_name (int = 0) {} // without any arguments
```

PROGRAM 4

A program to demonstrate the default initialisation of the constructor member function of a class object of the students' information system such as name, roll number, gender, height and weight.

```
#include <iostream>
using namespace std;
class student {
        private :
                char name[20];
                long int rollno;
                char gender;
                float height;
                float weight;
        public :
                student();          // constructor
                void display();
};
student :: student()
{
        name[0] = '\0';
        rollno = 0;
        gender = '\0';
        height = 0;
        weight = 0;
}
void student :: display()
{
        cout << " name= " << name <<endl ;
        cout << " rollno = " << rollno <<endl;
        cout << " gender= " <<gender<< endl;
        cout << " height = " << height << endl;
        cout << " weight = " << weight << endl;
}
int main()
{
        student a;
        cout << " demonstration of default constructor \n";
        a.display();
        return 0;
}
```

Output

```
demonstration of default constructor
name=
roll no = 0
gender=
height   = 0
weight= 0
```

PROGRAM 5

A program to demonstrate the default initialisation of the constructor member function of a class object where the default constructor is created by the compiler automatically when the default constructor is not defined by the user.

```
#include <iostream>
using namespace std;
class student {
        private :
                char name[20];
                long int rollno;
                char gender;
                float height;
                float weight;
        public :
                void display();
};
void student :: display()
{
        cout << “ name= ” << name <<endl ;
        cout << “ rollno = ” << rollno <<endl;
        cout << “ gender= ” << gender << endl;
        cout << “ height = ” << height << endl;
         cout << “ weight = ” << weight << endl;
}
int main()
{
        student a;
        a.display();
        return 0;
}
```

Output

```
name = �������
rollno = 1108544020
gender = X
height = 3.98791e-34
weight = 36.7598
```

Overloading Constructors

The overloading constructor is a concept in OOPs in which the same constructor name is called with different arguments. Depending upon the type of argument, the constructor will be invoked automatically by the compiler to intialise the objects.

PROGRAM 6

A program to demonstrate how to define, declare and use the overloading of constructors to initialise the objects for different datatypes.

```
#include <iostream>
using namespace std;
class abc {
        public:
        abc();
        abc(int);
        abc(float);
        abc(int,  float);
};
abc :: abc()
{
        cout <<“calling default constructor \n”;
}
abc :: abc (int a)
{
        cout << “\n calling constructor with int \n”;
        cout << “ a = ” << a << endl;
}
abc :: abc (float fa)
{
        cout <<“\n calling constructor with floating point number \n”;
        cout <<“ fa = ” << fa << endl;
}
abc :: abc(int a, float fa)
{
        cout << “ \n calling constructor with int and float \n”;
        cout << “ a = ” << a << endl;
        cout << “ fa = ” << fa << endl;
}
int main()
{
        abc obj;
        obj(10);
        obj(1.1f);
        obj(20,-2.2);
        return 0;
}
```

Output

```
calling default constructor
calling constructor with int
a = 10
calling constructor with floating point number
constructor with int and float
a = 20
fa = -2.2
```

Constructors in Nested Classes

A class within a class is called as nested class. The scope rules to access the nested class constructor is the same as that of the member functions of the nested classes. The scope resolution operator (::) is used to identify the outer and inner class objects and the constructor member functions.

PROGRAM 7

```
#include <iostream>
using namespace std;
class abc {
        public:
                abc();
                class x {
                        public:
                                x();
                };
};
abc:: abc()
{
        cout << “abc - class constructor \n”;
}
abc::x :: x()
{
        cout << “x - class constructor \n”;
}
int main()
{
        abc obj;
        abc::x obj2;
        return 0;
}
```

Output

```
abc - class constructor
x - class constructor
```

Destructor

A destructor is a function that automatically executes when an object is destroyed. A destructor function gets executed whenever an instance of the class to which it belongs goes out of existence. The primary usage of the destructor function is to release space on the heap. A destructor function may be invoked explicitly.

Syntax Rules for Writing a Destructor Function. The rules for writing a destructor function are:

- A destructor function name is the same as that of the class it belongs except that the first character of the name must be a tilde (~).
- It is declared with no return types (not even void) since it cannot ever return a value.
- It cannot be declared static, const or volatile.
- It takes no arguments and therefore cannot be overloaded.
- It should have public access in the class declaration. The general syntax of the destructor function in C++ is,

```
class class_name {
      private :
                  // data variables
                  // methods
            protected :
                  // data
      public :
                  class_name();    // constructor
                  ~class_name(); // destructor
                  // other methods
};
```

PROGRAM 8

A program to simulate a simple banking system in which the initial balance and the rate of interest are read from the keyboard and these values are initialised using the constructor member function. The destructor member function is defined in this program to destroy the class objects created using the constructor member function.

The program consists of the following methods:

∑ *To initialise the balance and the rate of interest using the constructor member function.*
∑ *To make a deposit.*
∑ *To withdraw an amount from the balance.*
∑ *To find the compound interest based on the rate of interest.*
∑ *To know the balance amount.*
∑ *To display the menu options.*
∑ *To destroy the object of class, the destructor member function is defined.*

```
#include <iostream>
using namespace std;
class account {
            private :
                  float balance;
                  float rate;
            public :
                  account();            // constructor
                  ~account();           // destructor
                  void deposit ();
                  void withdraw ();
                  void compound();
                  void getbalance();
                  void menu();
}; //end of class definition
account :: account() // constructor
{
            cout << “ enter the initial balance \n”;
            cin >> balance;
            cout << “ interest rate \n”;
            cin >> rate;
}
```

```
account :: ~account() // constructor
{
        cout << " data base has been deleted \n";
}
//deposit
void account :: deposit ()
{
        float amount;
        cout << " enter the amount";
        cin >> amount;
        balance = balance+amount;
}
//attempt to withdraw
void account :: withdraw ()
{
        float amount;
        cout << " how much to withdraw ? \n";
        cin >> amount;
        if (amount <= balance) {
                balance = balance-amount;
                cout << " amount drawn = " <<amount << endl;
                cout << " current balance = " << balance << endl;
        }
        else
                cout << 0;
}
void account :: compound ()
{
        float interest;
        interest = balance*rate;
        balance = balance+interest;
        cout << "interest amount = " << interest <<endl;
        cout << " total amount = " << balance << endl;
}
void account :: getbalance()
{
        cout << " Current balance = " ;  cout << balance << endl;
}
void account :: menu()
{
        cout << " d -> deposit\n";
        cout << " w -> withdraw\n";
        cout << " c -> compound interest\n";
        cout << " g -> get balance\n";
        cout << " m -> menu \n";
        cout << " q -> quit \n";
        cout << " option, please ? \n";
}
int main()
{
        account acct;
        char ch;
        acct.menu();
        while ( (ch = cin.get()) != 'q'){
                switch (ch) {
                                case ‘d’ :
                                        acct.deposit();
                                        break;
                                case ‘w’ :
                                        acct.withdraw();
                                        break;
                                case ‘c’ :
                                        acct.compound();
                                        break;
                                case ‘g’ :
                                        acct.getbalance();
                                        break;
                                case ‘m’:
                                        acct.menu();
                                        break;
                                } // end of switch statement
                        }
                cout << endl;
                return 0;
}
```

Output

```
enter the initial balance  1000
interest rate  0.2
d -> deposit
w -> withdraw
c -> compound interest
g -> get balance
m -> menu
q -> quit
option, please?
d
enter the amount  500
g
Current balance = 1500
w
how much to withdraw?
1000
amount drawn = 1000
Current balance = 500
c
interest amount = 100
total amount = 600
q
data base has been deleted
```

INLINE MEMBER FUNCTIONS

The keyword inline is used as a function specifier only in function declarations. The inline specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation.

PROGRAM 9

```
#include <iostream>
using namespace std;
class sample {
```

```
            private :
                        int x;
            public :
                        void getdata();
                        void display();
};
inline void sample:: getdata()
{
            cout << “ enter a number ? \n”;
            cin >> x;
}
inline void sample :: display()
{
            cout << “ entered number is = “ << x ;
            cout << endl;
}
int main()
{
            sample obj;
            obj.getdata();
            obj.display();
            return 0;
}
```

Output

```
enter a number?  10
entered number is = 10
```

Static Variables

Main characteristic of the static variables is that the static variables are automatically initialised to zero unless it has been initialised by some other value explicitly.

PROGRAM 10

```
#include <iostream>
using namespace std;
class sample {
            private :
                        static int counter;
            public :
                        void display();
};
int sample::counter = 100;
void sample:: display()
{
            int i;
            for (i = 0; i <= 10; ++i) {
                        counter = counter+i;
            }
            cout << “ sum of the counter value = “ << counter;
            cout << endl;
}
int main()
{
            sample obj;
            int i;
            for ( i = 0; i < 5; ++i) {
                  cout << " count = " << i+1 << endl;
                  obj.display();
                  cout << endl;
          }
          return 0;
}
```
Output

```
Count = 1
sum of the counter value = 155
count =2
sum of the counter value = 210
count = 3
sum of the counter value = 265
count = 4
sum of the counter value = 320
count = 5
sum of the counter value = 375
```

Static Member Functions

The keyword static is used to precede the member function to make a member function static. The static function is a member function of class and the static member function can manipulate only on static data member of the class. The static member function acts as global for members of its class without affecting the rest of the program.
The purpose of static member is to reduce the need for global variables by providing alternatives that are local to a class. A static member function is not part of objects of a class.
A static or nonstatic member function cannot have the same name and the same arguments type. And further, it cannot be declared with the keyword const. The static member function is instance dependent, it can be called directly by using the class name and the scope resolution operator. If it is declared and defined in a class, the keyword static should be used only on the declaration part.

PROGRAM 11

A program to check how many instances of the class object are created using the static member function.

```
//both static data member and static function member
#include <iostream>
using namespace std;
class sample {
      private :
            static int count; // static data member declaration
      public :
            sample();
            static void display(); // static member function
      };
//static data definition
int sample :: count = 0;
sample :: sample ()
{
      ++count;
}
void sample :: display()
{
      cout << " counter value = " << count << endl;
}
int main()
{
      cout << " before instantiation of the object " << endl;
      sample::display();
      sample obj1,obj2,obj3,obj4;
      cout << " after instantiation of the object " << endl;
```

```
        sample::display();
        return 0;
}
```
Output

```
before instantiation of the object
counter value = 0
after instantiation of the object
counter value = 4
```

FRIEND FUNCTIONS

The main concepts of the object-oriented programming paradigm are data hiding and data encapsulation. Whenever data variables are declared in a private category of a class, these members are restricted from accessing by non-member functions. The private data values can be neither read nor written by non-member functions. If any attempt is made directly to access these members, the compiler will display an error message as "inaccessible data type". The best way to access a private data member by a non-member function is to change a private data member to a public group. When the private or protected data member is changed to a public category, it violates the whole concept of data hiding and data encapsulation.

To solve this problem, a friend function can be declared to have access to these data members. Friend is a special mechanism for letting non-member functions access private data. A friend function may be either declared or defined within the scope of a class definition. The keyword friend inform the compiler that it is not a member function of the class. If the friend function is declared within the class, it must be defineded outside the class, but should not be repeated the keyword friend.

The general syntax of the friend function is,

friend return_type function_name (parameters);

A friend declaration is valid only within or outside the class definition.

1. The following is a valid program segment shows how a friend function is defined within the scope of a class definition

```
                class alpha {
            private:
                int x;
            public :
                void getdata();
                friend void display (alpha abc)
                {
                    cout << " value of x = " << abc.x;
                    cout << endl;
                }
    }; // end of class definition
```

2. The following program segment shows how a friend function is defined out of the class definition

```
      class alpha {
        private:
            int x;
          public :
            void getdata();
            friend void display (alpha abc);
    }; // end of class definition
    void display(alpha abc)   //non-member function without scope::operator
    {
        cout << " value of x = " << abc.x;
        cout << endl;
    }
```

The following is an invalid declaration of the friend function. The keyword friend should not be repeated in both the function declaration and definition.

```
class alpha {
    private:
        int x;
    public :
        void getdata();
```

```
         friend void display (alpha abc);
    };
    friend void display(alpha abc) //the keyword friend is repeated
    {
        cout << " value of x = " << abc.x;
        cout << endl;
    }
```
The friend declaration is unaffected by its location in the class. The C++ compiler permits the declaration of a friend function either in a public or a private section, which does not affect its access right.

The following declarations of a friend function are valid:

3. The friend function disp () is declared in the public group
```
     class rst {
         private :
             int x;
         public :
             void getdata();
             friend void disp();
     };
```
4. The friend function disp () is declared in the private group
```
    class second {
        private :
            int x;
            friend void disp();
        public :
            void getdata();
};
```

PROGRAM 12
```
//declaring friend function
#include <iostream>
using namespace std;
class sample {
    private :
        int x;
    public :
        void getdata();
        friend void display(class sample);
};
void sample :: getdata()
{
    cout << " enter a value for x \n";
    cin >> x;
}
void display(class sample abc)
{
    cout << " Entered number is :" << abc.x;
    cout << endl;
}
int main()
{
    sample obj;
    obj.getdata();
    cout <<" accessing the private data by non-member function \n";
    display(obj);
    return 0;
}
```
Output
```
enter a value for x
```

```
10099
accessing the private data by non-member function
Entered number is: 100
```

DYNAMIC MEMORY ALLOCATIONS

Two operators, namely, new and delete are used in dynamic memory allocations which are described in detail in this section.

(a) New The new operator is used to create a heap memory space for an object of a class. C++ provides a new way in which dynamic memory is allocated. In reality, the new keyword calls upon the function operator new() to obtain storage.
Basically, an allocation expression must carry out the following three things:

(i) Find storage for the object to be created,
(ii) Initialise that object, and
(iii) Return a suitable pointer type to the object.

The new operator returns a pointer to the object created.
The general syntax of the new operator is,

```
data_type pointer = new data_type;
```

where data_type can be a short integer, float, char, array or even class objects.
For example,

```
new int; // an expression to allocate a single integer
 new float; // an expression to allocate a floating value
```

If the call to the new operator is successful, it returns a pointer to the space that is allocated. Otherwise it returns the address zero if the space could not be found or if some kind of error is detected.

(b) Delete The delete operator is used to destroy the variable space which has been created by using the new operator dynamically. In reality, the delete keyword calls upon the function operator delete() to release storage which was created using the new operator.
The general syntax of the delete operator is:

```
delete pointer ;
```

As the new operator returns a pointer to the object being created, the delete operator must define a only pointer name but not with data type. The delete operator takes no arguments for deleting a single instance of a memory variable created by a new operator.
For example,

```
char *ptr_ch = new char; // memory for a character is allocated
int *ptr_i = new int; // memory for an integer is allocated
delete ptr_ch; // delete memory space
 delete ptr_i; // delete
```

Note that the delete operator is used for only releasing the heap memory space which was allocated by the new operator. If attempts are made to release memory space using delete operator that was not allocated by the new operator, then it gives unpredictable results. The following usage of the delete operator is invalid, as the delete operator should not be used twice to destroy the same pointer.

```
char *ptr_ch = new char; // allocating space for a character
delete ptr_ch;
delete ptr_ch;     // error
```

PROGRAM 13

A program to create a dynamic memory allocation for the standard data types: integer, floating point, character and double. The pointer variables are initialised with some data and the contents of the pointers are displayed on the screen.

```
//using new and delete operators
#include <iostream>
using namespace std;
int main()
{
            int *ptr_i = new int (25);
            float *ptr_f = new float(-10.1234F);
            char *ptr_c = new char('a');
            double *ptr_d = new double (1234.5667L);
            cout << "contents of the pointers " << endl;
            cout << "integer = " << *ptr_i << endl;
            cout << " floating point value = " << *ptr_f << endl;
            cout << “char = “ << *ptr_c << endl;
            cout << “double = “ << *ptr_d << endl;
            delete ptr_i;
            delete ptr_f;
            delete ptr_c;
            delete ptr_d;
            return 0;
}
```

Output

```
contents of the pointers
integer = 25
floating point value = -10.1234
char = a
double = 1234.57
```

(c) Array Data Type When an object is an array data type, a pointer to its initial element is returned.

```
 new int;
 new int[20];
```

The general syntax of the new operator for the array data type is,

```
 data_type pointer = new data_type[size];
```

where data_type can be a short integer, float, char, array or even class objects, and size is the maximum number of elements that are to be accommodated.

(d) Use of New Operator to Allocate Memory for a Two-Dimensional Array A two-dimensional array can be declared using the new operator as,

```
 new int [10][20];
```

The general syntax of the new operator for the two-dimensional array data type is,

```
data_type (pointer)[size] = new data_type[size][size];
```

where data_type can be a short integer, float, char, array or even class objects and size is the maximum number of elements that are to be accommodated.

PROGRAM 14

A program to allocate contiguous memory space for an array of integers using the new operator and the
object of the array is destroyed by the delete operator. This program reads a set of numbers from the keyboard and displays it on the screen.

```
//using new and delete operators for array data type
#include <iostream>
using namespace std;
int main()
{
            int *ptr_a = new int[20];
            int *ptr_n = new int;
            cout << “ how many numbers are there ? \n”;
            cin >> *ptr_n;
            for ( int i = 0; i<= *ptr_n -1; ++i) {
                        cout << “ element a[“ << i <<“] = “;
                        cin >> ptr_a[i];
            }
            cout << “ contents of the array \n”;
            for (int i = 0; i<= *ptr_n -1; ++i) {
                        cout << ptr_a[i] ;
                        cout << ‘\t’;
            }
            delete ptr_n;
            delete [] ptr_a;
           return 0;
}
```

Output

```
how many numbers are there?
5
element a[0] = 11
element a[1] = 22
element a[2] = 33
element a[3] = 44
element a[4] = 55
contents of the array
11 22 33 44 55
```

THIS POINTER

The this pointer is a variable which is used to access the address of the class itself. Sometimes, the this pointer may have return data items to the caller. In other words, the this pointer is a pointer accessible only within the non-static member functions of a class, struct, or union type. It points to the object for which the member function is called.
Static member functions do not have a this pointer. *The general syntax of the this pointer is:*

```
this
this->class_member
(*this).class_member
```

For example, the following assignment statements are equivalent:

```
void Date::set Month( int mn ) {
    month = mn; // These three statements are equivalent
     this->month = mn;
        (*this).month = mn;
}
```

PROGRAM 15

A program to demonstrate how to use the this pointer for accessing the members of a class object.

```
#include <iostream>
using namespace std;
class date_info {
        public:
            int day,month,year;
            void setdate(int d,int m, int y);
            void display();
       };
            void date_info :: setdate(int d, int m, int y)
            {
                  //three assignment statements are equivalent
                  day = d;
                 this->month = m;
                 (*this).year = y;
             }
      void date_info :: display()
      {
            cout << “Today’s date is : “ << this->day << “/”;
            cout << (*this).month << “/” << this->year << ‘\n’;
          }
       int main()
       {
            date_info obj;
           obj.setdate(10,10,2007);
           obj.display();
          return 0;
       }
```

Output

```
Today’s date is: 10/10/2007
```

Inheritance

In order to maintain and reuse the class objects easily, it is required to relate disperate classes into another. This chapter presents the salient features of the inheritance that probably is the most powerful feature of the object-oriented programming. Inheritance is the process of creating new classes from an existing class. The existing class is known as the base class and the newly created class is called as a derived class.

The main advantages of the inheritance are:

- reusability of the code,
- to increase the reliability of the code, and
- to add some enhancements to the base class.

Type of Inheritance

C++ supports five types of inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

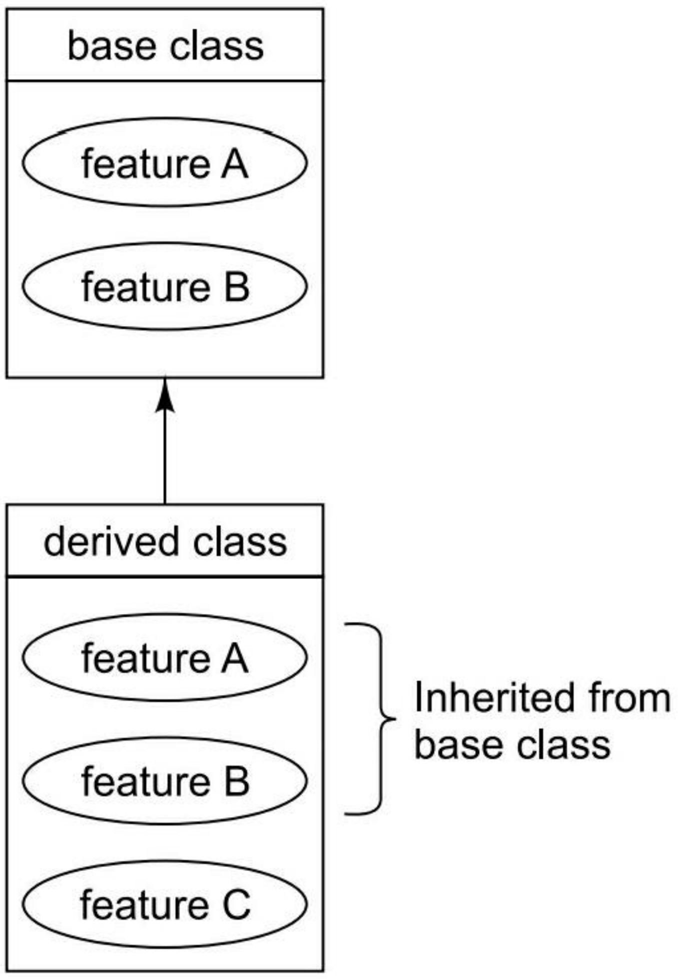


Fig. Single Inheritance

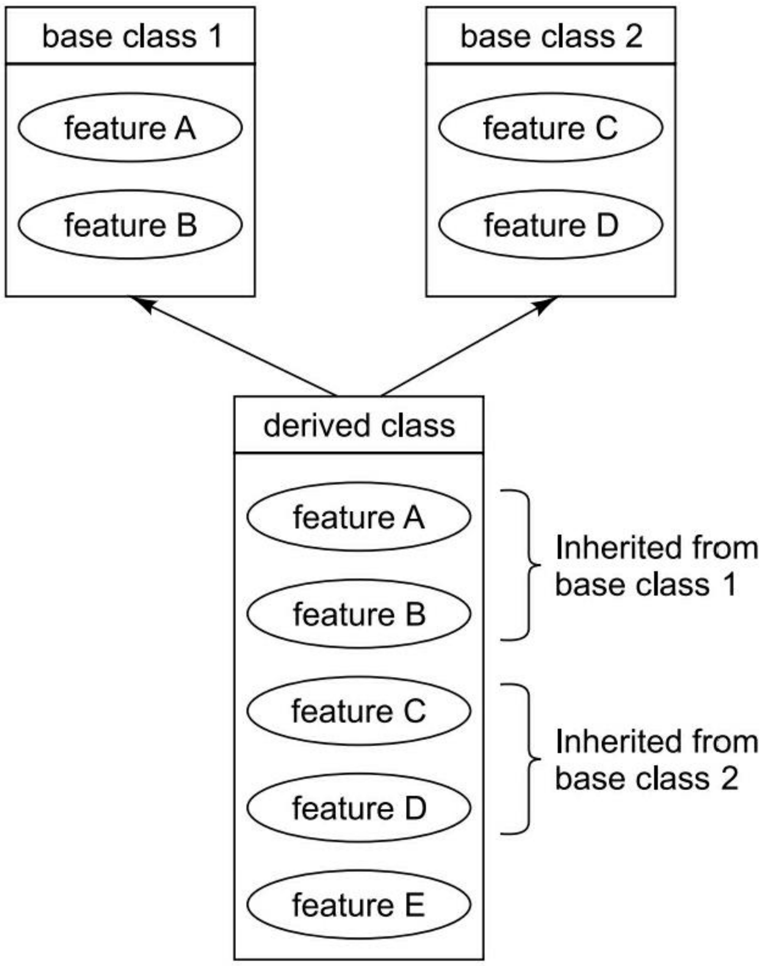


Fig. Multiple Inheritance

SINGLE INHERITANCE

Single inheritance is the process of creating new classes from an existing base class. The existing class is known as the direct base class and the newly created class is called as a singly derived class.

Single inheritance is the ability of a derived class to inherit the member functions and variables of the existing base class.

The general syntax of the derived class declaration is as follows.

```
class derived_class_name : private/public/protected base_class_name
{
    private :
        // data members
    public :
        // data members
        // methods
    protected :
        // data members
};
```

PROGRAM 1

A program to read the data members of a basic class such as name, roll number and gender from the keyboard and display the contents of the class on the screen. This program does not use any inheritance concepts.

```
#include <iostream>
#include <iomanip>
using namespace std;
class basic_info {
    private :
       char name[20];
       long int rollno;
       char gender;
    public :
       void getdata();
       void display();
     }; // end of class definition
void basic_info :: getdata()
{
     cout << " enter a name ? \n";
     cin >> name;
    cout << " roll no ? \n";
    cin >> rollno;
    cout << " Gender ? \n";
    cin >>gender;
}
void basic_info :: display()
{
 cout << name << " ";
 cout << rollno << " ";
 cout << gender << " ";
}
int main()
{
 basic_info a;
 cout << "enter the following information \n";
 a.getdata();
 cout << " —————————————— \n";
 cout << " Name Roll Gender \n";
 cout << " —————————————— \n";
 a.display();
 cout << endl;
 cout << " —————————————— \n";
 return 0;
}
```

Output

```
enter the following information
enter a name?
Ravi
roll no?
20071
Gender?
M
----------------------------
Name Rollno Gender
----------------------------
Ravi   20071 M
----------------------------
```

PROGRAM 2

A program to read the derived class data members such as name, roll number, gender, height and weight from the keyboard and display the contents of the class on the screen. This program demonstrates a single inheritance concept which consists of a base class and a derived class.

```
//single inheritance
#include <iostream>
#include <iomanip>
using namespace std;
class basic_info {
    private :
       char name[20];
       long int rollno;
       char gender;
   public :
        void getdata();
        void display();
 }; // end of class definition
class physical_ t :public basic_info
{
    private :
       float height;
       float weight;
    public:
         void getdata();
         void display();
 }; //end of class definition
void basic_info :: getdata() {
      cout << " enter a name ? \n";
      cin >> name;
      cout << " roll no ? \n";
      cin >> rollno;
      cout << " Gender ? \n";
      cin >> gender;
 }
void basic_info :: display() {
    cout << name << " ";
    cout << rollno << " ";
    cout << gender << " "; }
void physical_ t :: getdata() {
    basic_info::getdata();
    cout << " height ? \n";
    cin >> height;
    cout << " weight ?\n";
    cin >> weight;
}
void physical_ t :: display() {
    basic_info::display();
    cout << height << " ";
    cout << weight << " ";
}
int main() {
   physical_ t a;
   cout << "enter the following information \n";
   a.getdata();
```

```
    cout << " ——————————————————————— \n";
    cout << " Name Rollno Gender Height Weight \n";
    cout << " ——————————————————————— \n";
    a.display();
    cout << endl;
    cout << " ———————————————————————  \n";
 return 0;
}
```

Output

```
enter the following information
enter a name?
Himanshu
roll no?
27003
Gender?
M
height?
179
weight?
78
-----------------------------------------
Name        Rollno   Gender     Height Weight
-----------------------------------------
Himanshu  27003     M            179      62
-----------------------------------------
```

TYPES OF DERIVATION

Inheritance is a process of creating a new class from an existing class. While deriving the new classes, the access control specifier gives the total control over the data members and methods of the base classes. A derived class can be defined with one of the access specifiers, viz. private, public and protected.

Public Inheritance

The most important type of access specifi er is public. In a public derivation,

- each public member in the base class is public in the derived class.
- each protected member in the base class is protected in the derived class.
- each private member in the base class remains private in the base class.

The general syntax of the public derivation is:

```
 class base_class_name {
 -------
 -------
 }
 class derived_class_name : public base_class_name
 {
 -------
 -------
 }
```

Private Inheritance

In a private derivation,

- each public member in the base class is private in the derived class.
- each protected member in the base class is private in the derived class.
- each private member in the base class remains private in the base class and hence it is visible only in the base class.

The general syntax of the private derivation is,

```
class base_class_name {
-------
-------
}
class derived_class_name : private base_class_name
{
-------
-------
}
```

Protected Inheritance

In a protected inheritance,

- each public member in the base class is protected in the derived class.
- each protected member in the base class is protected in the derived class.
- each private member in the base class remains private in the base class and hence it is visible only in the base class.

The general syntax of the protected derivation is,

```
class base_class_name {
-------
-------
}
class derived_class_name : protected base_class_name
{
-------
-------
}
```

AMBIGUITY IN SINGLE INHERITANCE

Derived classes, these names must be without ambiguity. The scope resolution operator (::) may be used to refer to any base member explicitly. This allows access to a name that has been redefined in the derived class. For example, the following program segment illustrates how ambiguity occurs when the getdata () member function is accessed from the main () program.

```
class baseA {
      public :
          int i;
          getdata();
};
class baseB {
      public :
           int i;
           getdata();
};
class derivedC : public baseA, public baseB {
     public :
         int i;
         getdata();
```

```
 }
 int main()
 {
      derivedC obj;
      obj.getdata();
      return 0;
 }
```
The members are ambiguous without scope operators. When the member function getdata () is accessed by the class object. Therefore it is essential to declare the scope operator explicitly to call a base class member as illustrated below:

```
 obj.baseA::getdata();
 obj.baseB::getdata();
```

PROGRAM 3

A program to demonstrate how ambiguity is avoided in single inheritance using scope resolution operator.

```
#include <iostream>
using namespace std;
class baseA {
       private :
           int i;
       public :
            void getdata(int x);
            void display();
};
class baseB {
      private :
          int j;
      public :
          void getdata(int y);
          void display();
};
class derivedC : public baseA,public baseB
{
};
void baseA :: getdata(int x)
{
      i = x;
}
void baseA :: display()
{
      cout << " value of i = " << i << endl;
}
void baseB :: getdata(int y)
{
      j = y;
}
void baseB :: display()
{
    cout << " value of j = " << j << endl;
}
int main()
{
      derivedC objc;
      int x, y;
      cout << " enter a value for i ?\n";
      cin >> x;
      objc.baseA::getdata(x);     // member is ambiguous without scope
      cout << " enter a value for j ?\n";
      cin >> y;
      objc.baseB::getdata(y);
      objc.baseA::display();
      objc.baseB::display();
      return 0;
}
```

Output

```
enter a value for i?
10
enter a value for j?
20
value of i = 10
value of j = 20
```

ARRAY OF CLASS OBJECTS AND SINGLE INHERITANCE

This section emphasises on how an array of class objects can be inherited from a base class. Once a derived class has been defined, the way of accessing a class member of the array of class objects are same as the ordinary class types.

The general syntax of the array of class objects in a singly derived class is:

```
class baseA {
    private :
        -------
        -------
    public :
        -------
        -------
};
class derivedB : public baseA {
    private :
        -------
        -------
    public :
        -------
        -------
};
int main()
{
    derivedB obj[100];      //array of class objects of the derived class
     -------
     -------
     return 0;
}
```

PROGRAM 4

```
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 100;
class basic_info {
      private :
         char name[20];
         long int rollno;
         char gender;
      public :
         void getdata();
        void display();
}; // end of class definition
class physical_ t :public basic_info
{
      private :
          float height;
          float weight;
      public:
          void getdata();
          void display();
}; //end of class definition
void basic_info :: getdata() {
      cout << " enter a name ? \n";
      cin >> name;
      cout << " roll no ? \n";
     cin >> rollno;
     cout << "Gender ? \n";
     cin >> gender;
}
void basic_info :: display() {
     cout<< setw(10) << name;
     cout << setw(10)<< rollno ;
     cout << setw(10)<< gender;
}
void physical_ t :: getdata() {
       basic_info::getdata();
       cout << " height ? \n";
       cin >> height;
       cout << " weight ?\n";
       cin >> weight;
}
void physical_ t :: display() {
      basic_info::display();
      cout << setw(10)<< height;
      cout << setw(10) << weight;
}
int main() {
     physical_ t a[MAX];
     int i,n;
     cout << " How many students ? \n";
     cin >> n;
     cout << "enter the following information \n";
     for (i = 0; i <= n-1; ++i) {
     cout << "record : " << i+1 << endl;
     a[i].getdata();
 }
    cout << endl;
    cout << " ------------------------------------ \n";
    cout << “Name     Rollno    Gender    Height     Weight   \n”;
    cout << “ ----------------------------------- \n”;
    for (i = 0; i <= n-1; ++i) {
       a[i].display();
       cout << endl;
    }
     cout << endl;
     cout << “ ------------------------------------ \n”;
     return 0;
}
```

MULTIPLE INHERITANCE

Multiple inheritance is the process of creating a new class from more than one base classes. The syntax for multiple inheritance is similar to that of single inheritance.

For example, *the following program segment shows how a multiple inheritance is defined:*

```
class baseA {
    -------
    -------
};
class baseB {
    -------
    -------
};
class C : public baseA,public baseB
{
    -------
    -------
};
```

The class C is derived from both classes baseA and baseB.

Multiple inheritance is a derived class declared to inherit properties of two or more parent classes (base classes). Multiple inheritance can combine the behaviour of multiple base classes in a single derived class.

The rules of inheritance and access do not change from a single to a multiple inheritance hierarchy. A derived class inherits data members and methods from all its base classes, regardless of whether the inheritance links are private, protected or public.

(1) Multiple inheritance with all public derivation.

```
class baseA { // base class 1
    -------
    -------
};
class baseB { // base class 2
    -------
    -------
};
class baseC { // base class 3
   -------
   -------
};
class derivedD : public baseA,public baseB,public baseC
{
    -------
    -------
};
```

(2) Multiple inheritance with all private derivation.

```
 class baseA { // base class 1
      -------
      -------
 };
 class baseB { // base class 2
      -------
      -------
 };
 class baseC { // base class 3
       -------
       -------
 };
class derivedD : private baseA,private baseB,private baseC
 {
 -------
 -------
 };
```

(3) Multiple inheritance with all mixed derivation.

```
 class baseA { // base class 1
       -------
       -------
 };
 class baseB { // base class 2
       -------
       -------
 };
 class baseC { // base class 3
       -------
       -------
 };
 class derivedD : private baseA,public baseB,private baseC
 {
      -------
      -------
 };
```

PROGRAM 5

```
#include <iostream>
using namespace std;
class basic_info {
      private :
            char name[20];
            long int rollno;
            char gender;
      public :
            void getdata();
            void display();
 }; // end of class definition
class academic_ t {
      private :
           char course[20];
           char semester[10];
           int rank;
      public :
        void getdata();
        void display();
}; // end of class definition
class financial_assit : private basic_info, private academic_ t
{
    private :
        float amount ;
    public :
        void getdata();
        void display();
}; // end of class definition
void basic_info :: getdata()
{
     cout << " enter a name ? \n";
     cin >> name;
     cout << " roll no ? \n";
     cin >> rollno;
     cout << " Gender ? \n";
     cin >> gender;
}
void basic_info :: display()
{
     cout << name << " ";
     cout << rollno << " ";
     cout << gender << " ";
}
void academic_ t :: getdata()
{
    cout << " course name ( BTech/MCA/DCA etc) ?\n";
    cin >> course;
    cout << " semester ( first/second etc)? \n";
    cin >> semester;
    cout << " rank of the student \n";
    cin >> rank;
}
void academic_ t :: display()
{
    cout << course << " " ;
    cout << semester << " ";
    cout << rank << " ";
}
void financial_assit :: getdata()
{
     basic_info:: getdata();
     academic_ t::getdata();
     cout << " amount in rupees ?\n";
     cin >> amount;
}
void financial_assit :: display()
{
     basic_info:: display();
     academic_ t::display();
     cout << amount << " ";
}
```

```
int main()
{
       financial_assit f;
       cout << “enter the following information for financial assistance\n”;
      f.getdata();
      cout << endl;
      cout << “ Academic Performance for Financial Assistance \n”;
      cout << “ ——————————————————————————————— \n”;
      cout << “ Name Rollno Gender Course Semester Rank Amount \n”;
      cout << “ ——————————————————————————————— \n”;
     f.display();
     cout << endl;
     cout << “ ——————————————————————————————— \n”;
     return 0;
}
```

Output

```
enter the following information for financial assistance
enter a name?
AjitKumar
roll no?
20071
Gender?
M
course name (BTech/MCA/DCA etc)?
MCA
semester ( first/second etc)?
First
rank of the student
3
amount in rupees?
10000
Academic Performance for Financial Assistance
---------------------------------------------------------------
 Name Rollno Gender Course Semester Rank Amount
---------------------------------------------------------------
 AjitKumar 20071 M MCA First 3 10000
---------------------------------------------------------------
```

Ambiguity in the Multiple Inheritance

To avoid ambiguity between the derived class and one of the base classes or between the base class themselves, it is better to use the scope resolution operator :: along with the data members and methods.

For example, *the following program segment illustrates how ambiguity occurs in both base classes and the derived class.*

```
#include <iostream>
using namespace std;
class baseA {
      public:
          int a;
};
class baseB {
      public:
         int a;
};
class baseC {
      public :
         int a;
};
class derivedD : public baseA,public baseB,public baseC
{
      public :
          int a;
};
int main()
{
       derivedD objd;
       objd.a = 10;            //local to the derived class
       objd.baseA::a = 20; //accessing the base class A member
       objd.baseB::a = 30; //accessing the base class B member
       objd.baseC::a = 40; // accessing the base class C member
       return 0;
}
```

PROGRAM 6

A program to demonstrate how ambiguity is avoided in multiple inheritance using scope resolution operator.

```
#include <iostream>
using namespace std;
class baseA {
      public:
          int a;
};
class baseB {
      public:
           int a;
};
class baseC {
      public :
          int a;
};
class derivedD : public baseA,public baseB,public baseC
{
       public :
           int a;
};
int main()
{
      derivedD objd;
      objd.a = 10;
      objd.baseA::a = 20;
      objd.baseB::a = 30;
      objd.baseC::a = 40;
       cout << " value of a in the derived class = " << objd.a << endl;
       cout << " value of a in baseA = " << objd.baseA::a << endl;
       cout << " value of a in baseB = " << objd.baseB::a << endl;
       cout << " value of a in baseC = " << objd.baseC::a << endl;
       cout << endl;
       return 0;
}
```

Output

```
value of a in the derived class = 10
value of a in baseA = 20
value of a in baseB = 30
value of a in baseC = 40
```

MEMBER ACCESS CONTROL

Access control and its mechanism to access the individual members of a class as well as the derived class are explained. It has already been stated that the access mechanism of the individual members of a class is based on the use of the keywords public, private and protected.

Accessing the Public Data

The public members of a class can be accessed by the following:

- Member functions of the class.
- Nonmember functions of the class.
- Member function of a friend class.
- Member function of a derived class if it has been derived publicly.

For example, *the following are valid declarations for accessing the public data member of a class:*

(1) The member function of the class can access the public data.

```
class sample {
      public :
          int value;
          void getdata();
          void display()
         {
             ++ value; // valid
          }
};
```

(2) The nonmember function of the class can access the public data.

```
class sample {
     public :
        int a;
};
int main()
{
      sample obj;
     obj.a++; // valid
}
```

(3) The member function of the derived class can access the public data of the base class if it has been derived publicly.

```
class base {
    public :
        int value;
        void getdata()
};
class derived : public base {
     public :
     void display()
     {
           ++value; // valid
     }
};
```

(4) The following declaration for accessing the public data member is invalid as it has been declared as private inheritance.

```
class base {
       public :
            int value;
            void getdata()
};
class derived : private base {
       public :
       void display()
      {
             ++value; // invalid
      }
};
```

Accessing the Private Data

The private members of a class can be accessed only by the following:

- The member function of the class, and
- The member functions of the friend class in which it is declared.

In other words, the public member function of a derived class cannot access the private data member of the base class irrespective of whether the derived class has been inherited publicly or privately.

For example, *in the following program the declaration for accessing the private data members of base class by the public members of the derived class is invalid:*

(1) A derived class with private derivation.

```
class baseA {
       private :
       int value;
};
class derivedB : private baseA {
       public :
       void f()
       {
            ++ value;      // error, baseA:: value is not accessible
        }
};
```

(2) Even if the derived class has been derived publicly from the base class, the public member of a derived class cannot access the private member of the base class.

```
class baseA {
      private :
      int value;
};
class derivedB : public baseA {
       public :
       void f()
       {
            ++ value; // error, baseA:: value is not accessible
       }
};
```

Note that a derived class public member function cannot access the private member of a base class irrespective of whether the derived class has been derived publicly or privately.

The following program shows an error message indicating that baseA::value is not accessible.

```
#include <iostream>
using namespace std;
class base {
       public :
          int value;
          inline void getdata()
          {
               cout << " enter a number " << endl;
               cin >> value;
          }
}; // end of class definition
class derived : private base {
      public :
           void display()
           {
                 ++value;
           }
};// end of class definition
int main()
{
      derived obj;
      obj.getdata();
      obj.display();
      cout << " value in derived class = " << obj.value;
      return 0;
}
```

Compile time error
void base :: getdata() is inaccessible due to private inheritance
int base :: value is inaccessible due to private inheritance.

Accessing the Protected Data

The protected data members of a class can be accessed by the following:

- The member function,
- The friends of the class in which it is declared, and
- The member functions of the derived class irrespective of whether the derived class has been derived privately or publicly.

For example, *the following program segments illustrate how a protected data member of a base is accessed by the public member function of the derived class:*

(1) The protected data member of a class can be accessed by the public member function even if it has been derived privately, provided the derived class have a direct base.

```
class baseA {
      protected :
           int value;
};
class derivedB : private baseA {
     public :
     void f()
     {
         ++ value; // valid
     }
};
```

(2) The protected data member of a class cannot be accessed by the public member function of the derivedD because the derivedD has not been derived from the direct base of baseA.

```
class baseA {
      protected :
      int value;
};
class derivedB : private baseA {
     -------
```

```
      -------
};
class derivedC : private derivedB {
      -------
      -------
};
class derivedD : private derivedC {
      public :
         void f()
        {
             ++ value; // error, baseA::value is not accessible
        }
};
```

(3) The protected data of the base class can access the public member function of a derived class via public derivation.

```
class baseA {
     protected :
     int value;
};
class derivedB : public baseA {
       public :
       void f()
       {
            ++ value; // valid
        }
 };
```

PROGRAM 7

A program to demonstrate how the protected data member of a base class is accessed by the public member function of the derived class if it has been derived privately

```
#include <iostream>
using namespace std;
class baseA{
      protected:
            int value;
      public:
           baseA ()
           {
                cout << " enter a number : ";
               cin >> value;
           }
}; // end of base declaration
class derivedB: private baseA {
     public:
           void display ()
           {
                ++value;
               cout << " content of value = " << value << endl;
           }
}; // end of derived class definition
int main()
{
      derivedB obj1;
      obj1.display();
     return 0;
}
```
Output

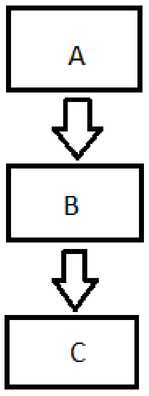
```
enter a number : 100
content of value = 101
```

SUMMARY OF THE INHERITANCE ACCESS SPECIFIER

Access specifier	**Accessible from own class**	**Accessible from derived calss**	**Accessible from objects outside class**
Public	Yes	Yes	Yes
Protected	Yes	Yes	NO
Private	Yes	No	No

Multilevel Inheritance

Multilevel inheritance is a process of driving a class from another derived class.



When one class inherits another class which is further inherited by another class, it is known as multilevel inheritance in C++. Inheritance is transitive so the last derived class acquires all the member of all its base classes.

A derived class with multilevel inheritance is declared as follow:

```
class A {
            ---------
            ---------
};
class B: public A {
            ---------
            ---------
};
class C: public B {
            ---------
            ---------
};
```
PROGRAM 8

```
#include<iostream>
using namespace std;
class student {
            protected:
```

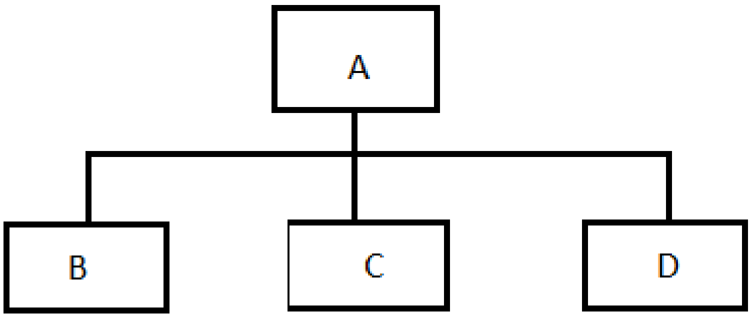
```
        int roll_number;
    public:
        void get_number(int);
        void put_number(void0;
};
void student :: get_number(int a)
{
    roll_number = a;
}
void student :: put_number()
{
    cout << “Roll Number:” << roll_number << endl;
}
class test : public student
{
    protected:
        float sub1;
        float sub2;
    public:
        void get_marks(float, float);
        void put_marks(void);
};
void test :: get_marks(float x, float y)
{
    sub1 = x;
    sub2 = y;
}
void test :: put_marks()
{
    cout << “Marks in sub1 =” << sub1 << endl;
    cout << “Marks in sub2 = “ << sub2<<endl;
}
class result : public test{
        float total;
    public:
        void display(void);
};
void result :: display(void)
{
    total = sub1 + sub2;
    put_number();
    put_marks();
    cout << “Total =” << total << endl;
}
int main()
{
    result student1;
    student1.get_number(111);
    student1.get_marks(75.0, 59.5);
    student1.display();
    return 0;
}
```
Output

```
Roll Number : 111
```

```
Marks in sub1 : 75
Marks in sub2 : 59.5
Total  = 134.5
```

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax:

```
class A {
            ---------
            ---------
};
class B: public A {
            ---------
            ---------
};
class C: public A {
            ---------
            ---------
};
class D: public A {
            ---------
            ---------
};
```

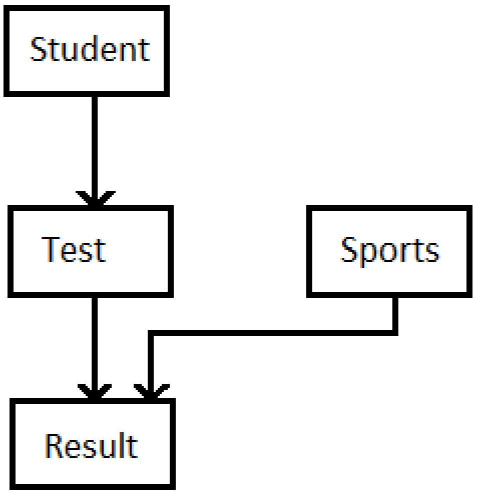
PROGRAM 9

```
#include<iostream>
using namespace std;
class shape{
            public:
                        int a;
                        int b;
                        void get_data(int n, int m)
                        {
                                    a = n;
                                    b = m;
                        }
};
class Rectangle : public shape{
            public:
                        int rect_area()
                        {
                                    int result = a*b;
```

```
                                        return result;
                        }
};
class Triangle : public shape
```

C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



POLYMORPHISM

The word ‘poly’ originated from a Greek word meaning many and ‘morphism’ from a Greek word meaning form, and thus ‘polymorphism’ means many forms. In object-oriented programming, polymorphism refers to identically named methods (member functions) that have different behaviour depending on the type of object they refer.

Polymorphism is the process of defining a number of objects of different classes into a group and call the methods to carry out the operation of the objects using different function calls. In other words, polymorphism means ‘to carry out different processing steps by functions having same messages’. It treats objects of related classes in a generic manner. The keyword virtual is used to perform the polymorphism concept in C++. Polymorphism refers to the run-time binding to a pointer to a method.

EARLY BINDING

Choosing a function in normal way, during compilation time is called as early binding or static binding or static linkage. During compilation time, the C++ compiler determines which function is used based on the parameters passed to the function or the function’s return type. The compiler then substitutes the correct function for each invocation. Such compiler-based substitutions are called static linkage.

By default, C++ follows early binding. With early binding, one can achieve greater efficiency. Function calls are faster in this case because all the information necessary to call the function are hard coded.

PROGRAM 1

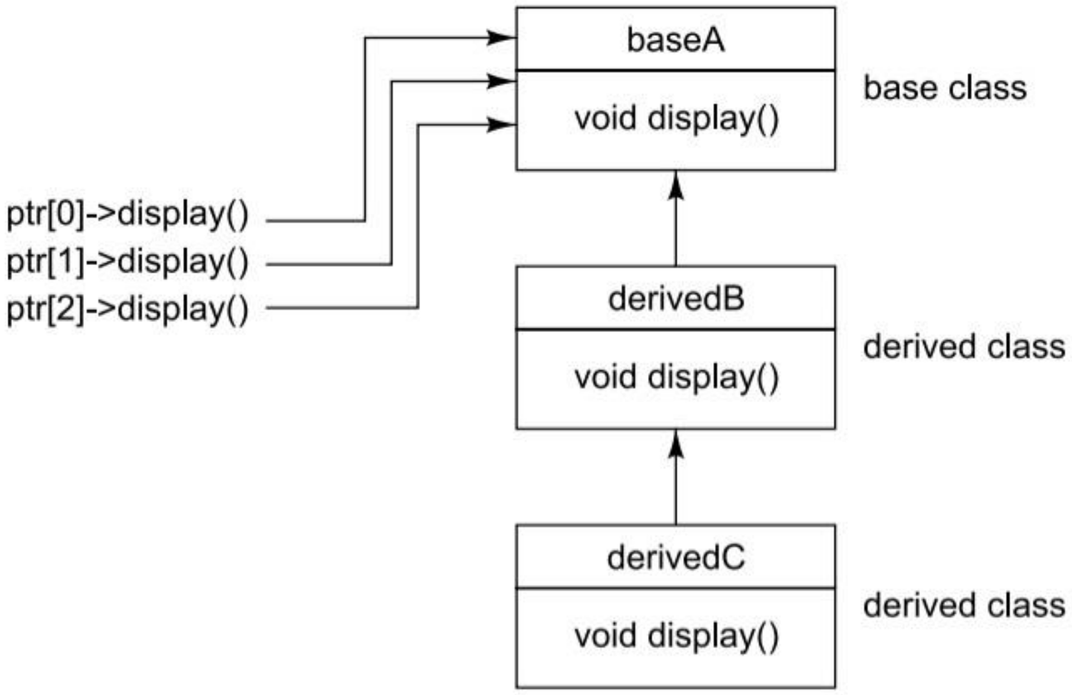
A program to demonstrate the compile time binding of the member functions of the class. The same message is given to access the derived class member functions from the array of pointers. As functions are declared as non-virtual, the C++ compiler invokes only the static binding.

```
//nonvirtual function
//demonstration of compile time binding using
//array of pointers
#include <iostream>
using namespace std;
class baseA {
        public :
                void display() {
                        cout << “One \n”;
                }
};
class derivedB: public baseA
{
         public:
                void display(){
                        cout << “ Two \n”;
                }
};
class derivedC: public derivedB
{
         public:
                void display(){
                        cout << “ Three \n”;
                }
};
int main()
{
        //define three objects
        baseA obja;
        derivedB objb;
        derivedC objc;
        baseA *ptr[3];        //define an array of pointers to baseA
        ptr[0] = &obja;
        ptr[1] = &objb;
        ptr[2] = &objc;
        for (int i = 0; i<=2; i++)
                ptr[i]->display();        //same message for all objects
        return 0;
}
```

Output

```
One
One
One
```

The function call through static binding



POLYMORPHISM WITH POINTERS

Pointers are also central to polymorphism in C++. To enable polymorphism, C++ allows a pointer in a base class to point to either a base class object or to any derived class object.

The following program segment illustrates how a pointer is assigned to point to the object of the derived class.

```
class baseA {
        -------
        -------
};
class derivedD : public baseA
{
        -------
        -------
};
int main()
{
        baseA *ptr;     // pointer to baseA
        derivedD objd;
        ptr = &objd;      //indirect reference objd to the pointer
        -------
        -------
        return 0;
}
```

The pointer ptr points to an object of the derived class objd.
By contrast, a pointer to a derived class object may not point to a base class object without explicit casting.
For example, *the following assignment statements are invalid:*

```
int main()
{
        baseA obja
        derivedD *ptr;
        ptr = &obja;        // invalid
        -------
        -------
        return 0;
}
```

Note that a derived class pointer cannot point to base class objects. But, the above code can be corrected by using explicit casting.

```
int main()
{
        square sqobj;
        rectangle *ptr;     //pointer of the derived class
        ptr = (rectangle*) &sqobj;      //explicit casting
        ptr->display();
        -------
        -------
        return 0;
}
```

VIRTUAL FUNCTIONS

A virtual function is one that does not really exist but it appears real in some parts of a program. Virtual functions are advanced features of the object-oriented programming concept and they are not necessary for each C++ program.
The general syntax of the virtual function declaration is:

```
class user_defined_name {
        private:
                -------
                -------
        public:
                virtual return_type function_name1(arguments);
                virtual return_type function_name2(arguments);
                virtual return_type function_name3(arguments);
                -------
                -------
};
```

To make a member function virtual, the keyword virtual is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual should be preceded by a return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration.

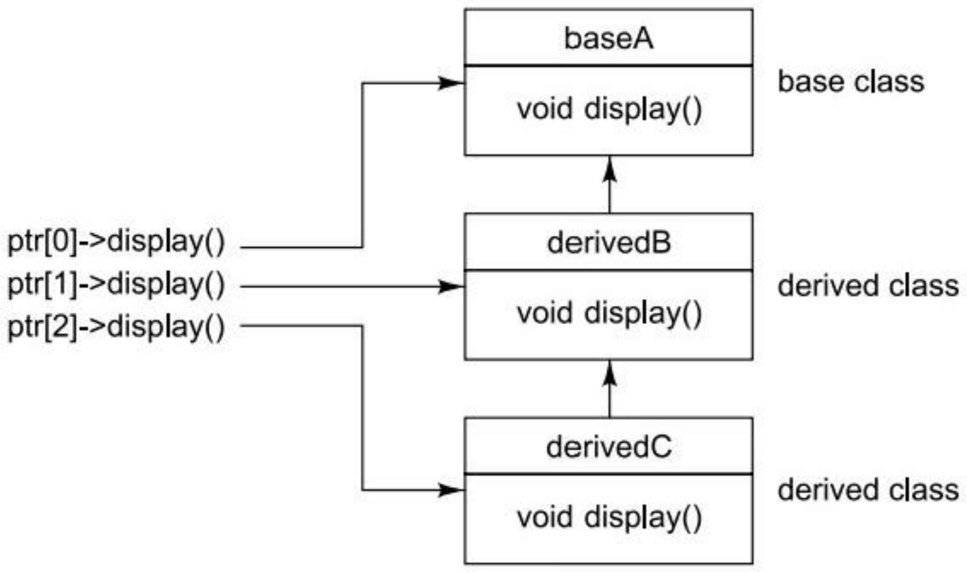
LATE BINDING

Choosing functions during execution time is called late binding or dynamic binding or dynamic linkage. Late binding requires some overhead but provides increased power and fl exibility. The late binding is implemented through virtual functions. An object of a class must be declared either as a pointer to a class or a reference to a class.
For example, *the following declaration of the virtual function shows how a late binding or run-time binding can be carried out:*

```
 class sample {
            private:
                        int x;
                        float y;
            public:
                        virtual void display();
                        int sum();
 };
 class derivedD: public baseA
 {
            private:
                        int x;
                        float y;
            public:
                        void display();     //virtual
                        int sum();
 };
 int main()
 {
            baseA *ptr;
            derivedD objd;
            ptr = &objd;
            -------
            -------
            ptr-> display();      //run time binding
            ptr-> sum();      // compile time binding
            return 0;
}
```

PROGRAM 2

A program to demonstrate the run-time binding of the member functions of a class. The same message is given to access the derived class member functions from the array of pointers. As functions are declared as virtual, the C++ compiler invokes the dynamic binding.

```
//virtual function
//demonstration of run time binding using
//array of pointers
#include <iostream>
using namespace std;
class baseA {
            public:
                        virtual void display() {
                                    cout << " One \n";
                        }
};
class derivedB : public baseA
{
            public:
                        virtual void display(){
                                    cout << " Two \n";
                        }
};
class derivedC : public derivedB
{
            public:
                        virtual void display(){
                                cout << " Three \n";
                        }
};
int main()
{
 //define three objects
            baseA obja;
            derivedB objb;
            derivedC objc;
            baseA *ptr[3];      //define an array of pointers to baseA
            ptr[0] = &obja;
            ptr[1] = &objb;
            ptr[2] = &objc;
            for (int i = 0; i<=2; i++)
                        ptr[i]->display();     //same message for all objects
            return 0;
}
```
Output
```
One
Two
Three
```


PURE VIRTUAL FUNCTIONS
A pure virtual function is a type of function which has only a function declaration. It does not have the function definition.

The following program segment illustrates how to declare a pure virtual function:

Case 1
```
 //pure virtual functions
 #include <iostream>
 using namespace std;
 class base {
            private:
                        int x;
                        float y;
            public:
                        virtual void getdata();
                        virtual void display();
 };
 class derivedB: public base {
            -------
            -------
 };
 void base :: getdata() //pure virtual function defi nition
 {
            // empty statements
 }
 void base :: display() //pure virtual function defi nition
 {
            // empty statements
 }
```

PROGRAM 3

A program to demonstrate how a pure virtual function is defined, declared and invoked from the object of a derived class through the pointer of the base class.

```
//pure virtual functions
#include <iostream>
using namespace std;
class base {
            private:
                        int x;
                        float y;
            public:
                        virtual void getdata();
                        virtual void display();
};
class derivedB: public base {
            private:
                        long int rollno;
                        char name[20];
            public:
                        void getdata();
                        void display();
};
void base :: getdata() { }    //pure virtual function
void base :: display() { }    //pure virtual function
void derivedB :: getdata()
{
            cout << " enter roll number of a student ? \n";
            cin >> rollno;
            cout << " enter name of a student ? \n";
            cin >> name;
}
void derivedB :: display()
{
            cout << " Roll number student's name \n";
            cout << rollno << '\t' << name << endl;
}
int main()
{
            base *ptr;
            derivedB obj;
            ptr = &obj;
            ptr->getdata();
            ptr->display();
            return 0;
}
```
Output
```
enter roll number of a student?
20075
enter name of a student?
Antony
Roll number      student's name
20075            Antony
```
Case 2 A pure virtual function can also have the following format, when a virtual function is declared within the class declaration itself. The virtual function may be equated to zero if it does not have a function definition.
```
//pure virtual functions
 #include <iostream>
using namespace std;
 class base {
            private:
                        int x;
                        float y;
            public:
                        virtual void getdata() = 0;
                        virtual void display() = 0;
 };
 class derivedB: public base {
            -------
            -------
 };
```
PROGRAM 4

A program to illustrate how to declare a pure virtual function and equate it to zero as it does not have any function parts.
```
//pure virtual functions
#include <iostream>
using namespace std;
class base {
            private:
                        int x;
                        float y;
            public:
                        virtual inline void getdata() = 0;
                        virtual inline void display() = 0;
};
class derivedB: public base {
            private:
                        long int rollno;
                        char name[20];
            public:
                        void getdata();
                        void display();
};
void derivedB :: getdata()
{
            cout << " enter roll number of a student ? \n";
            cin >> rollno;
            cout << " enter name of a student ? \n";
            cin >> name;
}
void derivedB :: display()
{
            cout << " Roll number student's name \n";
            cout << rollno << '\t' << name << endl;
}
int main()
{
            base *ptr;
            derivedB obj;
            ptr = &obj;
            ptr->getdata();
            ptr->display();
            return 0;
}
```

Output

```
enter roll number of a student?
20076
enter name of a student?
Ratanakumar
Roll number     student's name
20076           Ratanakumar
```

ABSTRACT BASE CLASSES

A class which consists of pure virtual functions is called an abstract base class.

PROGRAM 5

A program to demonstrate how to defi ne an abstract base class with pure virtual functions in which the function defi nition part has been defined without any statement. The members of the derived class objects are accessed through the base class objects through pointer technique.

```
//demonstration of abstract base classes
#include <iostream>
using namespace std;
class base {
            private:
                        int x;
                        float y;
            public:
 virtual void getdata();
 virtual void display();
};
class derivedB: public base {
            private:
                        long int rollno;
                        char name[20];
            public:
                        void getdata();
                        void display();
};
class derivedC: public base {
            private:
                        float height;
                        float weight;
```

```
        public:
                void getdata();
                void display();
};
void base :: getdata() {} //pure virtual function
void base :: display() {} // pure virtual function
void derivedB :: getdata()
{
        cout << “ enter a roll number of student ? \n”;
        cin >> rollno;
        cout << “ enter a name of student ? \n”;
        cin >> name;
}
void derivedB :: display()
{
        cout << “ Roll number student’s name \n”;
        cout << rollno << ‘\t’ << name << endl;
}
void derivedC :: getdata()
{
        cout << “ enter height of student ? \n”;
        cin >> height;
        cout << “ enter weight of student ? \n”;
        cin >> weight;
}
void derivedC :: display()
{
        cout << “ Height and weight of the student’s \n”;
        cout << height << ‘\t’ << weight << endl;
}
int main()
{
        base *ptr[3];
        derivedB objb;
        derivedC objc;
        ptr[0] = &objb;
        ptr[1] = &objc;
        ptr[0]->getdata();
        ptr[1]->getdata();
        ptr[0]->display();
        ptr[1]->display();
        return 0;
}
```

Output

```
enter a roll number of student?
20075
enter a name of student?
Ram
enter height of student?
167.56
enter weight of student?
64
Roll number    student’s name
20075          Ram
```

```
Height and weight of the student's
167.56   64
```

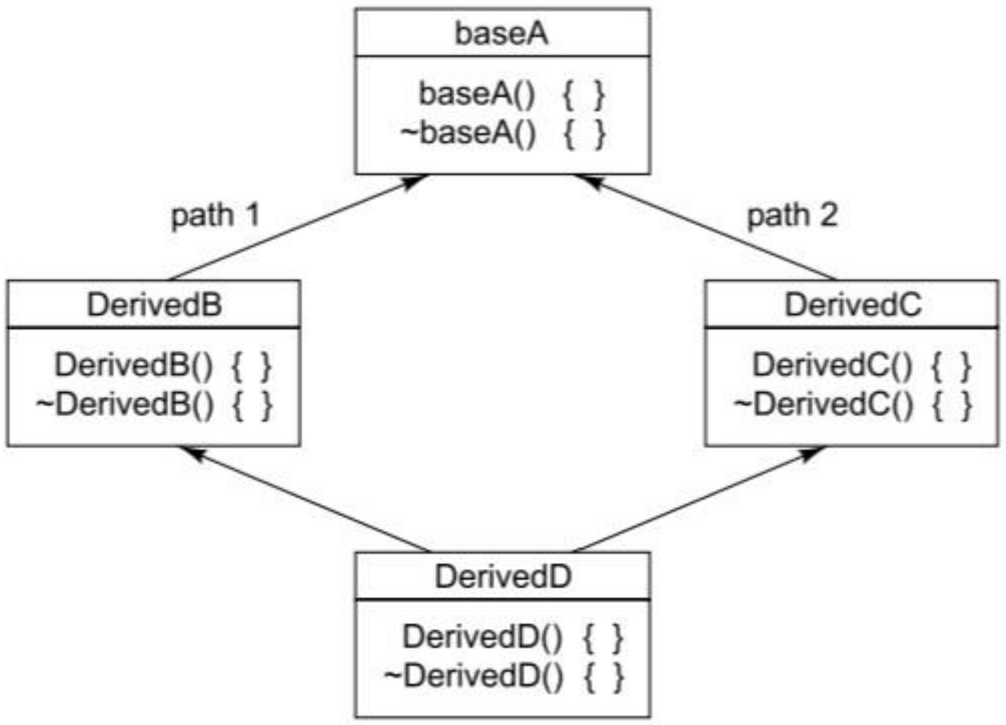
VIRTUAL BASE CLASSES

Inheritance is a process of creating a new class which is derived from more than one base classes. Multiple inheritance hierarchies can be complex, which may lead to a situation in which a derived class inherits multiple times from the same indirect base class.

For example, the following program segment illustrates how a base class can be derived twice from the derived class in different way.

```
class baseA {
        protected:
                int x;
                -------
                -------
};
class derivedB: public baseA { //path 1, through derivedB
        protected:
                -------
                -------
};
class derivedD: public baseA { //path 2, through derivedD
        protected:
                -------
                -------
};
class abc : public derivedB,public derivedD
{
        //the data member x comes twice
        -------
        -------
};
```

The data member x is inherited twice in the derived class abc, once through the derived class derivedB and again through derivedC. This is wasteful and confusing.



The above multiple repetition of the data member can be corrected by changing the derived class DerivedB and DerivedD into virtual base classes. Any base class which is declared using the keyword virtual is called a virtual base class. Virtual base classes are useful method to avoid unnecessary repetition of the same data member in the multiple inheritance hierarchies.

For example, *the following program segment illustrates how a base class is derived only once from the derived classes via virtual base classes.*

```
class baseA {
        protected:
                int x;
                -------
                -------
};
class derivedB: public virtual baseA { //path 1, through derivedB
        protected:
                -------
                -------
};
class derivedD: public virtual baseA { //path 2, through derivedD
        protected:
                -------
                -------
};
class abc : public derivedB,public derivedD
{
        //the data member x comes only once
        -------
        -------
};
```

PROGRAM 6

A program to define multiple derived classes which are accessing the same base class data members through indirect base references.

```
//using nonvirtual base classes
#include <iostream>
using namespace std;
class baseA {
        protected:
                int x;
        public:
                void getdata();
                void display();
};
class derivedB : public baseA {         //path 1, through derivedB
        protected:
                float y;
        public:
                void getdata();
                void display();
};
class derivedD : public baseA {         //path 2, through derivedD
        protected:
                char name[20];
        public:
                void getdata();
                void display();
};
class abc : public derivedB,public derivedD
{
        public:
                void getdata();
                void display();
};
void baseA :: getdata()
{
        cout << " enter an integer \n";
        cin >> x;
}
void baseA :: display()
{
        cout << " integer : " << x << endl;
}
void derivedB :: getdata()
{
        baseA::getdata();
        cout << " enter a floating point value \n";
        cin >> y;
}
void derivedB :: display()
{
        baseA :: display();
        cout << " real number :" << y << endl;
}
void derivedD :: getdata()
```

```
{
        baseA :: getdata();
        cout << " enter a string \n";
        cin >> name;
}
void derivedD :: display()
{
        baseA:: display();
        cout << " string : " << name << endl;
}
void abc:: getdata()
{
        derivedB :: getdata();
        derivedD :: getdata();
}
void abc:: display()
{
        derivedB :: display();
        derivedD :: display();
}
int main()
{
        abc obj;
        obj.getdata();
        obj.display();
        return 0;
}
```

Output

```
enter an integer
10
enter a floating point value
-2.2
enter an integer
20
enter a string
C++, world
integer : 10
real number :-2.2
integer : 20
string : C++,world
```

The protected data member x is accessed twice by the two derived classes through derivedB and derivedD. The content of the data variable is copied into two memory variables in the heap. It is certainly a repetition with different data and confusing if it is intended to process the same item of the base class.

PROGRAM 7

A program to define multiple derived classes which are accessing the same base class data members through indirect base references.

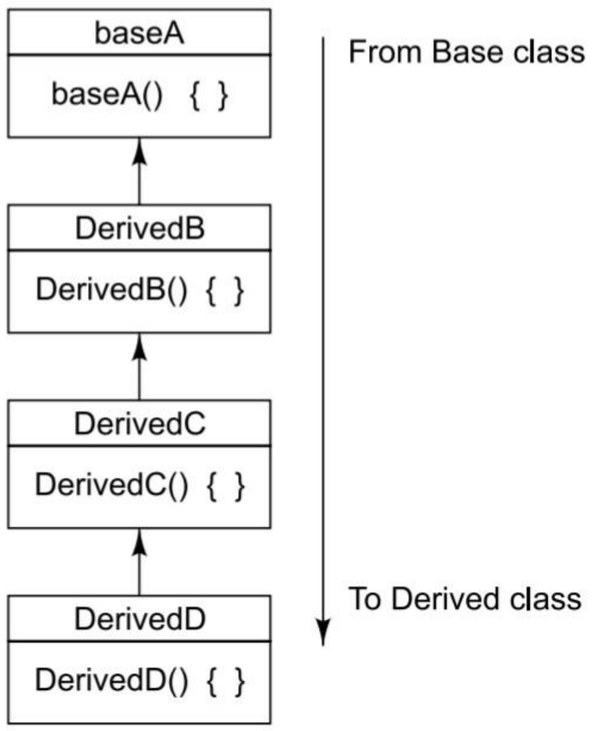
```
//using nonvirtual base classes
#include <iostream>
using namespace std;
class baseA {
        protected:
                int x;
        public:
                void getdata();
                void display();
};
class derivedB : public virtual baseA {         //path 1, through derivedB
        protected:
                float y;
        public:
                void getdata();
                void display();
};
class derivedD : public virtual baseA {         //path 2, through derivedD
        protected:
                char name[20];
        public:
                void getdata();
                void display();
};
class abc : public derivedB,public derivedD
{
        public:
                void getdata();
                void display();
};
void baseA :: getdata()
{
        cout << " enter an integer \n";
        cin >> x;
}
void baseA :: display()
{
        cout << " integer : " << x << endl;
}
void derivedB :: getdata()
{
        baseA::getdata();
        cout << " enter a floating point value \n";
        cin >> y;
}
void derivedB :: display()
{
        baseA :: display();
        cout << " real number :" << y << endl;
}
void derivedD :: getdata()
{
        baseA :: getdata();
        cout << " enter a string \n";
        cin >> name;
}
void derivedD :: display()
{
        baseA:: display();
        cout << “ string : ” << name << endl;
}
void abc:: getdata()
{
        derivedB :: getdata();
        derivedD :: getdata();
}
void abc:: display()
{
        derivedB :: display();
        derivedD :: display();
}
int main()
{
        abc obj;
        obj.getdata();
        obj.display();
        return 0;
}
```

Output

```
enter an integer
10
enter a floating point value
-2.2
enter an integer
20
enter a string
C++, world
integer : 20
real number :-2.2
integer : 20
string : C++,world
```

CONSTRUCTORS UNDER INHERITANCE

It has already been pointed out that whenever an object of a class is created, a constructor member function is invoked automatically and when an object of the derived class is created, the constructor for that object is called. This is due to the object of the derived class which contains the members of the base class also. Since the base class is also part of the derived class, it is not logical to call the constructors of the base class.



PROGRAM 8

A program to demonstrate how to defi ne and declare a constructor member function in the base class as well as in the derived class under the inheritance mechanism.

```
//constructors under inheritance
#include <iostream>
using namespace std;
class baseA {
        public:
                baseA(); //constructor
};
class derivedB: public baseA
        public :
                derivedB(); //constructor
};
baseA :: baseA()
{
        cout << “ base class constructor \n”;
}
derivedB :: derivedB()
{
        cout << “ derived class constructor \n”;
}
int main()
{
        derivedB objb;
        return 0;
}
```

Output

```
base class constructor
derived class constructor
```

DESTRUCTORS UNDER INHERITANCE

It has been seen that destructor is a special member function. It is invoked automatically to free the memory space which was allocated by the constructor functions. Whenever an object of the class is getting destroyed, the destructors are used to free the heap area so that the free memory space may be used subsequently.

PROGRAM 9

A program to demonstrate how the destructor member function gets fi red from the derived class objects to the base class objects through pointers.

```
//destructors under inheritance
#include <iostream>
using namespace std;
class baseA {
        public:
                ~baseA(); //destructor
};
class derivedB: public baseA {
        public:
                ~derivedB(); //destructor
};
baseA :: ~baseA()
{
        cout << " base class destructor\n";
}
derivedB :: ~derivedB()
{
        cout << " derivedB class destructor \n";
}
int main()
{
        derivedB objb;
        return 0;
}
```

Output

```
derivedB class destructor
base class destructor
```

PROGRAM 10

A program to display the message of both constructors and destructors of a base class and a derived class.

```
//destructor under inheritance
#include <iostream>
using namespace std;
class baseA {
        public:
                baseA() {       //baseA's constructor
                        cout << " base class constructor\n";
                }
                ~baseA() {      //baseA's destructor
                        cout << " base class destructor\n";
                }
};
class derivedD: public baseA {
        public:
                derivedD() {        //derivedD’s constructor
                        cout << “ derived class constructor\n”;
                }
                ~derivedD() {        //derivedD’s destructor
                        cout << “ derived class destructor\n”;
                }
};
int main()
{
        derivedD obj;
        return 0;
}
```

Output

```
base class constructor
derived class constructor
derived class destructor
base class destructor
```

FUNCTION OVERLOADING

Other than the concept of classes and objects, overloading of functions and operators are the most noteworthy features of C++.

Function overloading is a logical method of calling several functions with different arguments and data types that perform basically identical things by the same name. *The main advantages of using function overloading are:*

- eliminating the use of different function names for the same operation.
- helps to understand, debug and grasp easily.
- easy maintainability of the code.
- better understanding of the relation between the program and the outside world.

Function overloading is an easy concept in C++ and generally it is used within the class concept for member functions and constructors. The compiler classifies the overloaded function by its name and the number and type of arguments in the function declaration. The function declaration and definition is essential for each function with the same function name but with different arguments and data types.

Each function swap () must have a different definition such that the compiler can choose the correct function.

(1) A function for swapping two integers is,

```
void swap_int (int &a,int &b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
}
```

(2) A function for swapping two floating point numbers is,

```
void swap_ float ( float &a, float &b)
{
        float temp;
        temp = a;
        a = b;
        b = temp;
}
```

(3) A function for swapping two-character data types is,

```
void swap_char (char &a, char &b)
{
        char temp;
        temp = a;  a = b;
        b = temp;
}
```

PROGRAM 1

A program to demonstrate how function overloading is carried out for swapping of two variables of the various data types, namely integers, floating point numbers and character types.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
        void swap (int &ix, int &iy);
        void swap (float &fx, float &fy);
        void swap (char &cx, char &cy); // function declaration
        void swap (string &str1, string &str2);
        void menu();
        int ix, iy;
        float fx, fy;
        char cx, cy, ch;
        string str1, str2;
        menu ();
        while (( ch = cin.get()) != ‘q’)
        {
                switch(ch) {
                case ‘i’:
                        // swapping on integers
                        cout <<“enter any two integers \n”;  cin >> ix >> iy ;
                        cout <<“swapping of integers \n”;
                        cout <<“ix =“<< ix <<“iy =“<< iy << endl;
                        swap(ix,iy);
                        cout <<“after swapping \n”;
                        cout <<“ix =“<< ix <<“iy =“<< iy << endl;
                        break;
                case ‘f’:
                        //floating point numbers
                        cout <<“enter any two   floating point numbers\n”;
                        cin >> fx >> fy;
                        cout <<“swapping of floating point numbers \n”;
                        cout <<“fx =“<< fx <<“fy =“<< fy << endl;
                        swap(fx,fy);
                        cout <<“after swapping \n”;
                        cout <<“fx =“<< fx <<“fy =“<< fy << endl;
                        break;
                case ‘c’:
                        //swapping characters
                        cout <<“enter any two characters\n”;
                        cin >> cx >> cy;
                        cout <<“swapping of characters \n”;
                        cout <<“cx =“     << cx <<“cy =“<< cy << endl;
                        swap(cx,cy);
                        cout <<“after swapping \n”;
                        cout <<“cx =“<< cx <<“cy =“<< cy << endl;
                        break;
                case ‘s’:
                        //swapping strings
                        cout <<“enter any two strings\n”;
                        cin >> str1 >> str2;
                        cout <<“swapping of characters \n”;
                        cout <<“str1 =“<< str1 <<“str2 =“<< str2;
                        cout << endl;
                        swap(str1,str2);
                        cout <<“after swapping \n”;
                        cout <<“str1 =“<< str1 <<“str2 =“<< str2;
                        cout << endl;
                        break;
                case ‘m’:
```

```
                    menu();
                    break;
        }
        } //end of while statement
return 0;
}
void menu()
{
        cout <<“Swapping two data using function overloading\n”;
        cout <<“i -> integer swapping \n”;
        cout <<“f -> real number swapping \n”;
        cout <<“c -> character swapping \n”;
        cout <<“s -> string swapping \n”;
        cout <<“m -> menu \n”;
        cout <<“q -> quit \n”;
        cout <<“code, please ?\n”;
}
void swap (int &a,int &b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
}
void swap (float &a, float &b)
{
        float temp;
        temp = a;
        a = b;
        b = temp;
}
void swap (char &a, char &b)
{
        char temp;
        temp = a;
        a = b;
        b = temp;
}
void swap (string &a, string &b)
{
        string temp;
        temp = a;
        a = b;
        b = temp;
}
```
Output

```
Swapping two data using function overloading
i -> integer swapping
f -> real number swapping
c -> character swapping
s -> string swapping
m -> menu
q -> quit
```

```
code, please?
i
enter any two integers
100       200
swapping of integers
ix = 100 iy = 200
after swapping
ix = 200 iy = 100
f
enter any two floating point numbers
3.3       -5.6
swapping of floating point numbers
fx = 3.3 fy = -5.6
after swapping
fx = -5.6 fy = 3.3
c
enter any two characters
x         y
swapping of characters
cx = x cy = y
after swapping
cx = y cy = x
s
enter any two strings
C++       Java
swapping of characters
str1 = C++          str2 = Java
after swapping
str1 = Java         str2 = C++
q
```

PROGRAM 2

A program to find the square of a given number belonging to the three data types, namely integers, floating point and double precision numbers using overloading of functions

```
#include <iostream>
using namespace std;
class abc {
        public:
                int square (int);
                float square (float);
                double square (double);
                void menu();
        };
int abc :: square (int a)
{
        return(a*a);
}
float abc :: square (float a)
{
        return(a*a);
}
double abc :: square (double a)
```

```
{
        return(a*a);
}
void abc :: menu()
{
        cout << “Finding the square of a given number\n”;
        cout << “ i -> integers\n”;
        cout << “ f -> real numbers \n”;
        cout << “ d -> double precision numbers\n”;
        cout << “ m -> menu \n”;
        cout << “ q -> quit \n”;
        cout << “ code, please ?\n”;
}
int main()
{
        abc obj;
        int x, xsq;
        float y, ysq;
        double z, zsq;
        char ch;
        obj.menu();
        while (( ch = cin.get()) != ‘q’) {  switch (ch) {
                case ‘i’:
                        cout << “enter an integer ” << endl;
                        cin >> x;
                        xsq = obj.square (x);
                        cout << “ x = ” << x;
                        cout << “ and its square = ” << xsq << endl;
                        break;
                case ‘f’:
                        cout << “ enter a floating point number\n”;
                        cin >> y;
                        ysq = obj.square (y);
                        cout << “ y = ” << y ;
                        cout << “ and its square = ” << ysq <<endl;
                        break;
                case ‘d’:
                        cout << “ enter a double” << endl;
                        cin >> z;
                        zsq = obj.square (z);
                        cout << “ z = ” << z;
                        cout << “ and its square = ” << zsq << endl;
                        break;
                case ‘m’:
                        obj.menu();
                        break;
                }
        }
return 0;
}
```

Operator overloading

Operator overloading is another example of C++ polymorphism. In fact, some form of operator overloading is available in all high-level languages. For example, in BASIC the + operator can be used to carry out three different operations such as adding integers, adding two real numbers and concatenating two strings. Operator overloading is one of the most challenging and exciting features of C++. The main advantages of using overloading operators in a program are that it is much easier to read and debug.

The general syntax of operator overloading is,

return_type operator operator_to_be_overloaded (parameters);

Following are some examples for operator overloading of functions, their declaration and their equivalent conventional declaration.

(1) void operator++ ();   is equal to
    void increment ();
(2) int sum (int x, int y);   is equal to
    int operator+ (int x, int y);

Rules for overloading operators

Rule 1 Only those operators that are predefined in the C++ compiler can be used. Users cannot create new operators such as $, @ etc.

The following method of declaring an operator overloading is invalid

```
class newoperator {
        private:
                int x; int y;
        public :
                int operator $(); // error
                int operator @[]; // error
                int operator ::(); // error
};
```

Rule 2 Users cannot change operator templates. Each operator in C++ comes with its own template which defines certain aspects of its use, such as whether it is a binary operator or a unary operator and its order of precedence. This template is fixed and cannot be altered by overloading.

For example, ++ and – – cannot be used except in unary operators. During overloading, the prefixed incrementer/decrementer and the postfix incrementer/decrementer are not distinguished.

For an example,

```
++ operator ()
operator ++ ()
```

There is no difference between writing either prefix incrementer or postfix incrementer.

Rule 3 Overloading an operator never gives a meaning which is radically different from its natural meaning. For example, the operator * may be overloaded to add the objects of two classes but the code becomes unreadable.

```
// error class sample {
        private :
                int x; int y;
                public :
                        int operator *();          // adding two objects
                        int operator / ();         // subtracting two objects
};
int sample :: operator *()          // the code becomes unreadable
{
        return (x+y);
}
void main (void)
{ }
```

PROGRAM 3

A program to create a class of objects, namely, obja and objb. The contents of object obja is assigned to the object objb using the conventional assignment technique.

```
//using an assignment operator
//without overloading
#include<iostream>
```

```
using namespace std;
class sample {
        private :
                int x;
                float y;
        public :
                sample(int, float);
                void display();
};
sample :: sample ( int one, float two)
{
        x = one;
        y = two;
}
void sample :: display()
{
        cout << " integer number (x) = " << x << endl;
        cout << " floating value (y) = " << y << endl;
        cout << endl;
}
int main() {
        sample obj1(10, -22.55);
        sample obj2(20, -33.44);
        obj2 = obj1;
        cout << " contents of the first object \n";
        obj1.display();
        cout << " contents of the second object \n";
        obj2.display();
        return 0;
}
```

Output

```
contents of the first object
integer number (x) = 10
floating value (y) = -22.55

contents of the second object
integer number (x) = 10
floating value (y) = -22.55
```

PROGRAM 4

A program to create a class of objects, namely, obja and objb. The contents of object obja is assigned to the object objb using the operator overloading technique.

```
//overloading an assignment operator
#include<iostream>
using namespace std;
class sample {
        private :
                int x;
                float y;
        public :
                sample(int, float);
                 //overloading assignment operator
                void operator= (sample abc);
                void display ();
};
```

```
sample :: sample (int one, float two) {
        x = one;
        y = two;
}
void sample :: operator= (sample abc) {
        x = abc.x;
        y = abc.y;
}
void sample :: display() {
        cout << " integer number (x) = :" << x << endl;
        cout << " floating value (y) = :" << y << endl;
        cout << endl;
}
int main () {
        sample obj1(10, -22.55);
        sample obj2(20, -33.44);
        obj1.operator =(obj2);
        cout << "contents of the first object \n";
        obj1.display();
        cout << "contents of the second object \n";
        obj2.display();
        return 0;
}
```

Output

```
contents of the first object
integer number (x) = :20
floating value (y) = :-33.44

contents of the second object
integer number (x) = :20
floating value (y) = :-33.44
```

OVERLOADING OF BINARY OPERATORS

Binary operators overloaded by means of member functions take one formal argument which is the value to the right of the operator.

Overloading Arithmetic Operators

As arithmetic operators are binary operators, they require two operands to perform the operation. Whenever an arithmetic operator is used for overloading, the operator overloading function is invoked with single class objects.

PROGRAM 5

A program to perform overloading of a plus operator for finding the sum of the two given class objects.

```
//overloading arithmetic operators
#include<iostream>
using namespace std;
class sample {
        private :
                int value;
        public :
                sample ();
                sample (int one);
                sample operator+ (sample objb);
                void display();
};
sample ::sample () {
        value = 0;
}
sample :: sample (int one) {
	value = one;
}
sample sample :: operator+ (sample objb) {
	sample objsum;
	objsum.value = value+objb.value;
	return(objsum);
}
void sample :: display() {
	cout << “value = ” << value << endl;
}
int main() {
	sample obj1(10);
	sample obj2(20);
	sample objsum;
	objsum = obj1 + obj2;
	obj1.display();
	obj2.display();
	objsum.display();
	return 0;
}
```

Output

```
value = 10
value = 20
value = 30
```

Overloading of Comparison Operators

Comparison and logical operators are binary operators that require two objects to be compared and hence the result will be one of these:

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

The return value of the operator function is an integer. Operator overloading accepts an object on its right as a parameter and the object on the left is passed by this pointer

PROGRAM 6

A program to demonstrate how to overload a less than operator in a program to compare two classes of objects. The operator overloading returns either 1or 0 as the case may be.

```
//overloading comparison operators
#include<iostream>
using namespace std;
class sample {
	private:
		int value;
	public:
		sample ();
		sample (int one);
		void display ();
		int operator < (sample obj);
};
```

```
sample :: sample () {
            value = 0;
}
sample :: sample (int one) {
            value = one;
}
void sample :: display() {
            cout << " value " << value << endl;
}
int sample :: operator < (sample obja) {
            return (value < obja.value);
}
int main() {
            sample obja(20);
            sample objb(100);
            cout << (obja < objb) << endl;
            cout << (objb < obja) << endl;
            return 0;
}
```

Output

```
1
0
```

OVERLOADING OF UNARY OPERATORS

Overloading of incrementer and decrementer It is well known that C++ supports very unusual notation or operator that is used for incrementing and decrementing by 1. These operators can be used as either perfix or postfix. In general, overloading of these operators cannot be distinguished between prefix or postfix operation. However, whenever a postfix operation is overloaded, it takes a single argument along a member function of a class object.

The following program segment illustrates the overloading of an incrementer operator with prefix operation.

```
//overloading pre x ++ incrementer operator
#include<iostream>
using namespace std;
class fiibonacci {
            public:
                        void operator ++ ();
                        ------------
                        ------------
};
void fibonacci :: operator ++ ()
{
            -----------
            -----------
}
int main () {
            fibonacci obj;
            ---------
            ---------
            ++obj;
            return 0;
}
```

PROGRAM 7

A program to generate a Fibonacci series by overloading a prefix operator.

```
//overloading pre x ++ incrementer operator
#include <iostream>
```

```
#include<iomanip>
using namespace std;
class fibonacci {
        public:
                unsigned long int f0, f1, fib;
                fibonacci();        // constructor
                void operator++ ();
                void display ();
};
fibonacci :: fibonacci () {
        f0 = 0;
        f1 = 1;
        fib = f0+f1;
}
void fibonacci :: display() {
        cout << setw(4) <<fib;
}
void fibonacci :: operator++() {
        f0 = f1;
        f1 = fib;
        fib = f0+f1;
}
int main() {
        fibonacci obj;
        int n;
        cout << " How many fibonacci numbers are to be displayed ? \n";
        cin >>n;
        cout << obj.f0 << setw(4) << obj.f1;
        for (int i = 2; i <= n-1; ++i) {
                obj.display();
                ++obj;
        }
        cout << endl;
        return 0;
}
```

Output

```
How many fibonacci numbers are to be displayed?
8
0  1   1  2  3  5  8  13
```

FUNCTION TEMPLATE

Template is a method for writing a single function or class for a family of similar functions or classes in a generic manner. When a single function is written for a family of similar functions, it is called as a 'function template'. In this function at least one formal argument is generic.

In the previous section, we learn how a function can be overloaded for a similar type of operation to be executed and how these functions are defi ned and called in a program. In function overloading, though the same name is used for all functions which are defined as overloading, yet the codes are to be repeated for every function. Only the function names are same, but the function definition and declaration are repeated.

For example,

```
swap ( char *, char *) // swapping two character data types
{
        -------
        -------
}
swap ( int, int) // swapping two integer quantities
{
        -------
        -------
}
swap ( float, float) // swapping two floating point numbers
{
       -------
       -------
}
```

C++ provides certain features with the capability to define a single function for a group of similar functions. When a single function is written for a family of similar functions, they are called as function templates. The main advantage of using function template is avoiding unnecessary repetition of the source code. The object code becomes more compact and effi cient than the conventional way of declaring and defining the functions. Secondly, full data type checking is carried out.

The function template does not specify the actual data types of the arguments that a function accepts but it uses a generic or parameterised data type. In a function template at least one formal argument is generic.

The general syntax for declaring a function template in C++ is,

```
template<class T> T function_name (T formal arguments)
{
       -------
       -------
      return(T);
}
```

where the template and class are keywords in C++ and the function template must start with 'template' and the T is a parameterized data type.

The above declaration of a function template may be written in the following format also.

```
template<class T>
T function_name (T formal arguments)
{
        -------
        -------
 return(T);
}
```

PROGRAM 1

A program to defi ne the function template for swapping two items of the various data types, namely, integers, fl oating point numbers, characters and string based on object oriented programming approach.

```
#include <iostream>
using namespace std;
class sample {
      public:
          int a,b;
          float fa,fb;
          char ch1,ch2;
          string str1,str2;
          template <class T> void swap (T &a, T &b);
};
template <class T> void sample :: swap(T &a, T &b)
{
       T temp = a;
       a = b;
       b = temp;
}
int main()
{
         sample obj;
             cout <<“enter any two integers\n”;
              cin >> obj.a >> obj.b;
              cout <<“ Before swapping \n”;
              cout << “a = ” << obj.a << “ ,b = ” << obj.b << endl;
              obj.swap(obj.a,obj.b);
              cout <<“ After swapping \n”;
              cout << “a = ” << obj.a << “ ,b = ” << obj.b << endl;
             cout <<“enter two floating point numbers\n”;
              cin >> obj.fa >> obj.fb;
              cout <<“ Before swapping \n”;
              cout << “fa = ” << obj.fa << “ ,fb = ” << obj.fb << endl;
              obj.swap(obj.fa,obj.fb);
              cout <<“ After swapping \n”;
              cout << “fa = ” << obj.fa << “ ,fb = ” << obj.fb << endl;
             cout <<“enter any two characters\n”;
              cin.ignore();
              obj.ch1 = cin.get();
              cin.ignore();       //skip any whitespace like new line
              obj.ch2 = cin.get();
              cout <<“ Before swapping \n”;
              cout << “ ch1 = ” << obj.ch1;
              cout << “ ,ch2 = ” << obj.ch2;
              cout << endl;
              obj.swap(obj.ch1,obj.ch2);
              cout <<“ After swapping \n”;
              cout << “ ch1 = ” << obj.ch1;
             cout << “ ,ch2 = ” << obj.ch2;
              cout << endl;
             cout <<“enter any two words\n”;
              cin >> obj.str1;
              cin.ignore();      //skip whitespace, if any
              cin >> obj.str2;
```

```
            cout <<“ Before swapping \n”;
            cout << “str1 = ” << obj.str1 << “ ,str2 = ” << obj.str2;
            cout << endl;
            obj.swap(obj.str1,obj.str2);
            cout <<“ After swapping \n”;
            cout << “str1 = ” << obj.str1 << “ ,str2 = ” << obj.str2;
            cout << endl;
           return 0;
}
```

Output

```
enter any two integers
10 20
Before swapping
a = 10, b = 20
After swapping
a = 20, b = 10
enter two floating point numbers
1.1 2.2
Before swapping
fa = 1.1, fb = 2.2
After swapping
fa = 2.2, fb = 1.1
enter any two characters
a b
Before swapping
ch1 = a, ch2 = b
After swapping
ch1 = b, ch2 = a
enter any two words
Hyderabad Bangalore
Before swapping
str1 = Hyderabad, str2 = Bangalore
After swapping
str1 = Bangalore, str2 = Hyderabad
```

PROGRAM 2

A program to demonstrate how a function template is constructed to find square of a given number with different data types such as integers, floating point numbers and double.

```
//using function template
#include <iostream>
using namespace std;
template<class T> T square ( T n)
{
        return(n*n);
}
int main()
{
            int x,xsq;
            float y,ysq;
            double z,zsq;
            cout << “enter an integer \n”;
            cin >> x;
            cout << “ enter a floating point number ? \n”;
            cin >> y;
            cout << “ enter a double precision number \n”;
            cin >> z;
            xsq = square (x);
            cout << “ x = ” << x << “ and its square = ” << xsq << endl;
            ysq = square (y);
            cout << “ y = ” << y << “ and its square = ” << ysq << endl;
            zsq = square (z);
            cout << “ z = ” << z << “ and its square = ” << zsq << endl;
            return 0;
}
```

Output

```
enter an integer
2
enter a floating point number?
2.2
enter a double precision number
2.34544
x = 2 and its square = 4
y = 2.2 and its square = 4.84
z = 2.34544 and its square = 5.50111
```

CLASS TEMPLATE

In addition to function templates, C++ also supports the concept of class templates. By definition, a class template is a class defi nition that describes a family of related classes. C++ offers the user the ability to create a class that contains one or more data types that are generic or parameterised.

The manner of declaring the class template is the same as that of a function template. The keyword template must be inserted as a first word for defining a class template.

The general syntax of the class template is,

```
template <class T>
class user_defined_name
{
        private:
                -------
                -------
        public:
                -------
                -------
};
```

The following program segment illustrates how to define and declare a class template in C++.

```
#include <iostream>
using namespace std;
template <class T>
class sample {
        private:
                T value, value1, value2;
        public:
                void getdata();
                void sum();
};
int main()
{
        sample <int> obj1;
        sample < float> obj2;
        -------
```

```
            -------
            return 0;
}
```

PROGRAM 3

A program to illustrate how to define and declare a class template to find the sum of the given two data items.

```
//adding two parameterised data types
#include <iostream>
using namespace std;
template <class T>
class sample {
           private:
                       T value, value1, value2;
            public:
                       void getdata();
                       void sum();
};
template <class T>
void sample <T> :: getdata()
{
            cin >> value1 >> value2;
}
template <class T>
void sample <T> :: sum()
{
 T value;
            value = value1+value2;
            cout << " sum = " << value << endl;
}
int main()
{
            sample <int> obj1;
            sample < float> obj2;
            cout << " enter any two integers :" << endl;
            obj1.getdata();
            obj1.sum();
            cout << " enter any two floating point numbers :" << endl;
            obj2.getdata();
            obj2.sum();
 return 0;
}
```

Output

```
enter any two integers:
10   20
sum = 30
enter any two floating point numbers:
10.11   20.22
sum = 30.33
```

PROGRAM 4

A program to read a set of numbers from the user input and find the sum and average of given numbers using a class template.

```
// Finding sum and average of given numbers
#include <iostream>
using namespace std;
```

```
template <class T>
class sample{
        private:
                T a;
        public:
                void sum (int n);
};
template <class T>
void sample <T> :: sum(int n)
{
        T temp = 0;
        for (int i = 0; i <= n-1; ++i) {
                cout << “enter a value \n”;
                cin >> a;
                temp += a;
        }
        cout << “sum = ” << temp;
        T ave = temp/ n;
        cout << “ ,and Average = ” << ave << endl;
}
int main()
{
        int n;
        cout << “How many numbers ?\n”;
        cin >> n;
        cout <<“ for integers \n”;
        sample <int> obj1;
        obj1.sum(n);
        cout << “ floating point numbers\n”;
        sample <double> obj2;
        obj2.sum(n);
        return 0;
}
```

Output

```
How many numbers?
5
for integers
enter a value
1
enter a value
2
enter a value
3
enter a value
4
enter a value
5
sum = 15, and Average = 3
floating point numbers
enter a value
1.1
enter a value
2.2
enter a value
```

```
3.3
enter a value
4.4
enter a value
5.5
sum = 16.5, and Average = 3.3
```

OVERLOADING OF FUNCTION TEMPLATE

C++ also supports to construct a program for
overloading these function templates. One can implement the function templates for overloading either as a stand alone function template or member functions of a class.

PROGRAM 5

A program to demonstrate how to perform the function overloading using a function template.

```
//overloading of function template
#include <iostream>
using namespace std;
template <class T>
void display(T a)
{
            cout << “calling function template \n”;
            cout << “ a = ” << a << endl;
}
void display(int n)
{
            cout << “calling conventional display \n”;
            cout << “ n = ” << n << endl;
}
int main()
{
            display(10);
            display(1.1);
            return 0;
}
```

Output

```
calling conventional display
n = 10
calling function template
a = 1.1
```

PROGRAM 6

A program to demonstrate how to perform the function overloading using a function template. The function templates are defined as the member functions of a class.

```
//overloading of template methods
#include <iostream>
using namespace std;
template <class T>
class sample {
            public:
                        void display (T a);
                        void display (T a, T b);
                        void display (T a, T b, T c);
};
template <class T>
void sample <T> :: display (T a)
{
            cout << “ a = ” << a << endl;
}
template <class T>
void sample <T> :: display (T a, T b)
{
        cout << “ a = ” << a << “ b = ” << b << endl;
}
template <class T>
void sample <T> :: display (T a, T b, T c)
{
        cout << “ a = ” << a << “ b = ” << b << “ c = ” << c;
        cout << endl;
}
int main()
{
        sample <int> obj1;
        obj1.display(10);
        obj1.display(10,20);
        obj1.display(10,20,30);
        sample <double> obj2;
        obj2.display(1.1);
        obj2.display(1.1,2.2);
        obj2.display(1.1,2.2,3.3);
        return 0;
}
```

Output

```
a = 10
a = 10 b = 20
a = 10 b = 20 c = 30
a = 1.1
a = 1.1 b = 2.2
a = 1.1 b = 2.2 c = 3.3
```

PROGRAM 7

A program to demonstrate how to defi ne and declare a class template with a special member function, constructor and destructor.

```
//using class template
//defining constructor and destructor
#include <iostream>
using namespace std;
template < class T>
class sample {
        private :
                T value;
        public :
                sample ();
                ~sample ();
                inline void display ();
};
template <class T>
sample <T> :: sample()
{
        cout <<“constructor \n”;
}
template <class T>
sample <T> :: ~sample()
```

```
{
            cout << “destructor \n”;
}
template < class T>
void sample <T> :: display()
{
           cout <<“member function \n”;
}
int main()
{
            cout << “Calling class object for integer \n”;
            sample <int> obj1;
            obj1.display();
            cout << “Calling class object for floating point \n”;
            sample < float> obj2;
            obj2.display();
            return 0;
}
```

Output

```
Calling class object for integer
constructor
member function
Calling class object for floating point
constructor
member function
destructor
destructor
```

EXCEPTION HANDLING

An exception is an error or an unexpected event. The exception handler is a set of codes that executes when an exception occurs. Exception handling is one of the most recently added features and perhaps it may not be supported by much earlier versions of the C++ compilers. The main purpose of exception handling used in a program is to detect and manage runtime errors. The word exception comes from the exceptional program flow that occurs during a runtime error. C++ makes use of classes and objects in handling exceptions.

The ANSI/ISO C++ language defines a standard for exception handling. An exception handling is a type of error handling mechanism whenever a program encounters an abnormal situation or runtime errors. In order to implement and realise the exception handling, the following keywords are used in C++:

try
catch
throw

1. The Try Block

A `try` block protects any code within the block either directly or indirectly. A try block is like a sentinel that guards some section of code, shielding it from errors. Only the code inside a try block can detect or handle exceptions. The try block can also be nested like any other C++ code.

The general syntax of the try block is,

```
try
{
    /* the C++ code one wants to protect or shielding it from errors
    */
}
```

If an exception occurs, the program flow is interrupted.

2. The Throw Expression

The throw statement actually throws an exception of the specified type. When an exception occurs, the throw expression initialises a temporary object which is to match the type of argument it used in throw statement.

The general syntax of the throw statement is,

```
throw argument;
```

where `argument` is sent to the corresponding catch handler.

Different types of throw expression are given below:

(1) `throw " An error occurred ";`

This example specifies that an error message is to be displayed.

`throw object;`

This example specifies that object is to be passed on to a handler.

`throw;`

This example simply specifies that the last exception thrown is to be thrown again.

3. The Catch Block

The catch block receives the thrown exception. The exception handler is indicated by the catch keyword. The general syntax of the catch block is as follows:

```
catch (parameter)
{
    //handles error here
}
```

The Layout of Exception Handling

The general structure and layout of an exception handling is:

```
try
{
    // code to protect
}
catch (int x)
{
    // handler int errors
}
```

```
 catch (char *str)
 {
        // handler char * errors
 }
 catch (float dx)
 {
         // handler float errors
 }
```

This try block has three catch handlers. The first one that catches integer exceptions, the second one for character pointer and the third one catches float exceptions.

PROGRAM 1

A program to demonstrate how to detect a divide by zero error using the exception handling technique.

```
#include <iostream>
#include <string>
using namespace std;
class sample {
      private:
          float a, b;
      public:
          void getdata();
          void divide();
};
void sample ::getdata()
{
       cout<< "enter any two floating point numbers\n";
       cin >> a >> b;
}
void sample :: divide()
{
        string str;
        try {
            if (b == 0)
                  throw str;
            else
            {
                 float temp = a/b;
              cout << "Quotient = " << temp << endl;
            }
        }
       catch (string str) {
       cout << " Exception - Divide by zero \n";
      }
}
int main()
{
       sample obj;
       obj.getdata();
       obj.divide();
       return 0;
}
```

Output

```
enter any two floating point numbers
1  2
Quotient = 0.5
1  0
enter any two floating point numbers
```

Exception - Divide by zero

PROGRAM 2

A program to demonstrate how to define, declare and use an exception handling technique for detecting a memory out of range check error.

```
//memory out of range check

#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
class sample {
        private:
            int a[10];
            string str;
        public:
            void getdata();
            void display();
};
void sample :: getdata()
{
       try {
           for (int i = 0; i < 9; ++i) {
                 if (i > 5)
                       throw str;
                 else
                 {
                       cout <<"enter an element \n";
                           cin >> a[i];
                 }
            }
        }
        catch (string str) {
            cout << "\n Exception - Memory out of range";
       }
}
void sample ::display ()
{
          cout << "\n entered elements are \n";
          for (int i = 0; i <= 5; ++i)
                cout << setw(4) << a[i];
 }
int main()
{
         sample obj;
         obj.getdata();
         obj.display();
         return 0;
}
```

Output

```
enter an element
10
enter an element
20
enter an element
30
enter an element
40
enter an element
```

```
50
enter an element
60
Exception - Memory out of range
entered elements are
10  20  30  40  50  60
```

PROGRAM 3

A program to illustrate how to use the multiple catch statements for throwing the different types of data for handling exception.

```
#include <iostream>
using namespace std;
class sample {
       private:
           int a;
      public:
                  void getdata();
                  void display();
};
void sample :: getdata()
{
      cout << “enter anyone (0,1 or -1) \n”;
      cin >> a;
}
void sample :: display()
{
      int n;
      char ch;
      string str_name;
      try {
                  if (a == 0) throw n;
                  else if (a == 1) throw ch;
                  else if (a == -1) throw str_name;
       }
       catch (int n) {
             cout << “Exception - integer \n”;
     }
      catch (char ch) {
            cout << “Exception - Character \n”;
      }
       catch (string str_name) {
            cout << “Exception - String \n”;
       }
}
int main()
{
      sample obj;
      obj.getdata();
      obj.display();
      return 0;
}
```

Output

```
enter anyone (0,1 or -1)
0
Exception - integer
enter anyone (0,1 or -1)
1
Exception - Character
```

```
enter anyone (0,1 or -1)
-1
Exception - String
```

PROGRAM 4

A program to demonstrate how to implement catch all exceptions using the catch (...) statement.

//catch all exceptions

```
#include <iostream>
using namespace std;
class abc {
    public:
      void display(int x);
};
void abc :: display(int x)
{
   try
   {
            if (x == 0) throw ‘x’; // handling char
            if (x == 1) throw x; // handling int
            if (x == 2) throw 1.1f; // handling oat
            if (x == 3) throw 1000L; // handling long
    }
  catch (...)
  {
      cout << “Exception occurred \n”;
  }
}
int main()
{
    abc obj;
    obj.display(0);
  obj.display(1);
    obj.display(2);
    obj.display(3);
    return 0;
}
```

Output

```
Exception occurred
Exception occurred
Exception occurred
Exception occurred
```

PROGRAM 5

A program to illustrate how to realise rethrowing an exception in C++ using throw statement without any argument.

```
#include <iostream>
using namespace std;
class abc {
   public:
      void display(int x, int y);
};
void abc :: display(int x, int y)
{
   try
  {
      if (y == 0)
          throw y;
      else
          cout << “ x/y = ” << (double) x/ (double) y << endl;
  }
  catch (int)
```

```
    {
        cout << “Exception occurred \n”;
        throw;
    }
}
int main()
{
      abc obj;
      try {
          obj.display(1,0);
      }
       catch (int)
       {
            cout <<“Exception occurred in main \n”;
       }
        return 0;
}
```

Output

```
Exception occurred
Exception occurred in main
```

Standard Exceptions

The C++ library defines a number of common exceptions. The standard exceptions are defined in the following header file.

```
#include <stdexcept>
```

In general, exception handling mechanism can be classified into two categories, namely, logic error and runtime error.

(a) Logic Error The abnormal program termination can be caused due to the following error states:

- domain_error
- invalid_argument
- length_error
- out_of_range

(b) Runtime_error The run-time error can be occurred due to the following conditions.

- range_error
- overflow_error
- underflow_error

NAMESPACE

A namespace declaration identifi es and assigns a unique name to a user-declared namespace. Such namespaces are used to solve the problem of name collision in large programs and libraries. Programmers can use namespaces to develop new software components and libraries without causing naming conflicts with existing components. Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in “sub-scopes”, each one with its own name.

The general syntax of namespace declaration is, namespace identifier

```
      {
            // entities
      }
```

PROGRAM 1

A program to illustrate how to declare, define and realise a namespace mechanism.

```
#include <iostream>
using namespace std;
namespace A {
              int a = 10;
              void display();
}
```

```
namespace B {
            int a = 100;
            void display();
}
void A:: display()
{
            cout << “A :: display \n”;
            cout << A::a << endl;
}
void B :: display()
{
            cout << “B :: display\n”;
            cout << B::a << endl;
}
int main()
{
            A::display();
            B::display();
            return 0;
}
```

Output

```
A :: display
10
B :: display
100
```

Namespace Alias

A namespace alias definition substitutes a short name for a long and lengthy namespace name. The identifi er is a synonym for the qualifi ed namespace specifi er and becomes a namespace alias.

The namespace alias is declared in the following format:

```
namespace new_name = current_name;
```

For example,

```
namespace Very_Long_Namespace_Name
{
            int a;
            double x;
            void f();
}
namespace ABC = Very_Long_Namespace_Name
```

Now ABC is a namespace alias for Very_Long_Namespace_Name.

Nested Namespace

A namespace definition can be nested within another namespace definition. Every namespace definition must appear either at file scope or immediately within another namespace defi nition.

```
namespace outer
{
            entities_of_outer
            namespace inner
            {
                        entities_of_inner
                        namespace innermost
                        {
                                    entities_of_innermost
                        }
            }
}
```

PROGRAM 2

A program to declare, define and perform the nested namespace.

```
#include <iostream>
using namespace std;
namespace X {
        int a = 10;
        namespace Y {
                float a = 20.2f;
                namespace Z {
                        char a = ‘a’;
                }
        }
}
int main()
{
        cout << “X::a = ” << X::a << ‘\n’;
        cout << “X::Y::a = ” << X::Y::a << ‘\n’;
        cout << “X::Y::Z::a = ” << X::Y::Z::a << ‘\n’;
        return 0;
}
```

Output

```
X::a = 10
X::Y::a = 20.2
X::Y::Z::a = a
```

The Keyword Using

The keyword using is used to introduce a name from a namespace into the current declarative region.

The general syntax of the keyword using is given below:

```
 using userdefined_name :: member;
```

The following program segment shows how to use the keyword using:

```
namespace A
{
        int x = 5;
}
int main ()
{
        using A::x;
        cout << ++x << endl;
}
```

PROGRAM 3

A program to illustrate how to use the keyword using for namespace declaration.

```
// The keyword using
#include <iostream>
using namespace std;
namespace first
{
        int x = 5;
        int y = 10;
}
namespace second
{
        double x = 3.1416;
        double y = 2.7183;
}
```

```
int main ()
{
        using first::x;
        using second::y
        cout << x << endl;
        cout << y << endl;
        cout << first::y << endl;
        cout << second::x << endl;
        return 0;
}
```
Output

```
5
2.7183
10
3.1416
```

DATA FILE OPERATION (FILE HANDLING)

OPENING AND CLOSING OF FILES

File is a collection of data or a set of characters or may be a text or a program. Basically, there are two types of files in the C++: sequential files and random access files. The sequential fi les are very easy to create than random access fi les. In sequential fi les the data or text will be stored or read back sequentially. In random access files, data can be accessed and processed randomly.

This section describes how to open and close a data fi le using the header fi le <fstream>. The C++ Input and Output (I/O) class package handles fi le I/O as much as it handles standard input and output. The following methods are used in C++ to read and write files:

ifstream - to read a stream of object from a specifi ed file
ofstream - to write a stream of object on a specified file
fstream - both to read and write a stream of objects on a specifi ed file

The header file <fstream> is a new class which consists of basic file operation routines and functions. The fstream, ifstream and ofstream are called as derived class as these class objects are already defined in the basic input and output class namely <iostream>.

Opening a File

The following examples illustrate how fi les can be opened for reading and writing in C++. The member function open () is used to create a file pointer for opening a file in the disk.

ifstream The header file <ifstream> is a derived class from the base class of istream and is used to read a stream of objects from a file.

```
#include <fstream>
using namespace std;
int main()
{
        ifstream infile;
        infile.open(“data_file”); // opening a file
        -------
        -------
        return 0;
}
```

(b) **ofstream** The header file <ofstream> is derived from the base class of ostream and is used to write a stream of objects in a file.

```
#include <fstream>
using namespace std;
int main()
{
        ofstream infile;
        infile.open(“data_file”);
        -------
        -------
        return 0;
}
```

(c) fstream The header file <fstream> is a derived class from the base class of iostream and is used for both reading and writing a stream of objects on a file. The include <fstream> automatically includes the header file <iostream>. The following program segement shows how a file is opened for both reading and writing a class of stream objects from a specified file.

```
#include <fstream>
using namespace std;
int main()
{
        fstream infile;
        infile.open(“data_file”, ios::in | ios::out);
        -------
        -------
        return 0;
}
```

When a file is opened for both reading and writing, the I/O streams keep track of two file pointers — one for input operation and the other for output operation.

Note that for an instance istream (input), the default mode is ios:: in; for ofstream instance, the default mode is ios:: out. However, for an fstream (input/output) instance, there is no default mode. The bitwise OR operator is used to declare more than one mode.

Name of the Member function	Meaning
Ios::in	open a file for reading
ios::out	open a file for writing
ios::app	append at the end of a file
ios::ata	seek to the end of a file upon opening instead of beginning
ios:: trunc	delete a file if it exists and recreate it
ios:: nocreate	open a file if a file does not exist
ios:: replace	open a file if a file does exist
ios:: binary	open a file for binary mode; default is text

Closing a File

The member function close () is used to close a fi le which has been opened for file processing such as to read, to write and for both to read and write.

```
#include <fstream>
using namespace std;
int main()
{
        fstream infile;
        infile.open("data_file", ios::in | ios::out);
        -------
        -------
        infile.close (); // calling to close the le
        return 0;
}
```

STREAM STATE MEMBER FUNCTIONS

In C++, file stream classes inherit a stream state member from the ios class. The stream state member functions give the information status like end of file has been reached or file open failure and so on. The following stream state member functions are used for checking the open failure if any, when one attempts to open a fi le from the diskette.

eof()

fail()

bad()

good()

(a) eof() The eof() stream state member function is used to check whether a file pointer has reached the end of a file character or not. If it is successful, eof() member function returns a nonzero, otherwise returns 0.

The general syntax of the eof() stream state member function is:

```
#include <fstream>
using namespace std;
int main()
{
        ifstream infile;
        infile.open(“text”);
        while (!infile.eof()) {
                -------
                -------
        }
        return 0;
}
```

(b) fail() The fail() stream state member function is used to check whether a file has been opened for input or output successfully, or any invalid operations are attempted or there is an unrecoverable error. If it fails, it returns a nonzero character.

The general syntax of the fail () stream state member function is:

```
#include <fstream>
using namespace std;
int main()
{
        ifstream infile;
        in le.open(“text”);
        while (!infile.fail()) {
        cout << “ couldn’t open a file ” << endl;
        continue;
                -------
                -------
        }
        return 0;
}
```

(c) bad() The bad() stream state member function is used to check whether any invalid file operations has been attempted or there is an unrecoverable error. The bad() member function returns a nonzero if it is true; otherwise returns a zero.

The general syntax of the bad() stream state member function is:

```
#include <fstream>
#include <cstdlib>
using namespace std;
int main()
{
        ifstream infile;
        infile.open(“text”);
        if (infile.bad() ) {
                cerr << “ open failure ” << endl;
                exit(1);
        }
        -------
        -------
}
```

(d) **good()** The good() stream state member function is used to check whether the previous file operation has been successful or not. The good() returns a nonzero if all stream state bits are zero.

The general syntax of the good() stream state member function is:

```
#include <fstream>
#include <cstdlib>
using namespace std;
int main()
{
        ifstream infile;
        infile.open("text");
        if (infile.good()) {
                -------
                -------
        }
}
```

READING/WRITING A CHARACTER FROM A FILE

The following member functions are used for reading and writing a character from a specified file.

get()

put()

(a) **get()** The get() member function is used to read an alphanumeric character from a specified file.

The general syntax of the get() function is:

```
#include <fstream>
using namespace std;
int main()
{
        ifstream infile;
        char ch;
        infile.open ("text");
        -------
        -------
        while (( !infile.eof()) {
                ch = infile.get()
                -------
                -------
        } // end of while loop
}
```

(b) **put()** The put() member function is used to write a character to a specified file or a specified output stream.

The general syntax of the put() member function is:

```
#include <fstream>
using namespace std;
int main()
{
        ofstream outfile;
        char ch;
        outfile.open ("text");
```

```
                    -------
                    -------
                    while (( !ioutfile.eof()) {
                              ch = outfile.get()
                              cout.put(ch)       // display a character onto a screen
                              -------
                              -------
                    }
          }
```

PROGRAM 1

A program to demonstrate the writing of a set of lines to a specifi ed file, namely, “text.dat”.

```
//storing a text on a file
#include <fstream>
using namespace std;
int main()
{
          ofstream outfile;
          outfile.open(“text.dat”);
          outfile << “ this is a test \n”;
          outfile << “ program to store \n”;
          outfile << “ a set of lines onto a file \n”;
          outfile.close();
          return 0;
}
```

The contents of the “text.dat” file

```
this is a test
program to store
a set of lines onto a file
```

PROGRAM 2

A program to read a set of lines from the keyboard and to store it on a specified file.

```
//reading a text and store it on a specified file
#include <iostream>
#include <fstream>
using namespace std;
const int MAX = 2000;
int main()
{
          ofstream outfile;
          char fname[10],line[MAX];
          cout << “ enter a file name to be opened ?\n”;
          cin >> fname;
          outfile.open(fname);
          cout << “ enter a set of lines and terminate with @\n”;
          cin.get(line,MAX,’@’);
          cout << “ given input \n”;
          cout << line;
          cout << “ storing onto a file ....\n”;
          outfile << line;
          outfile.close();
          return 0;
}
```

Output

```
enter a file name to be opened?
 data
enter a set of lines and terminate with @
 this
 is a
 test program
 by Ravi
 @
given input
 this
 is a
 test program
 by Ravi
storing onto a file ....
```

PROGRAM 3

A program to demonstrate how to read a text file and to display the contents on the screen.

```
// reading and displaying a text le
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main()
{
        ifstream infile;
        char fname1[10];
        char ch;
        cout << " enter a file name ? \n";
       cin >> fname1;
       infile.open(fname1);
        if (infile.fail()) {
                cerr << " No such a file exists \n";
                exit(1);
        }
        while ( !infile.eof()) {
                ch = (char)infile.get();
                cout.put(ch);
        }
        infile.close();
        return 0;
}
```

PROGRAM 4

A program to copy the contents of a text file into another.

```
// file copy
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main() {
        ofstream outfile;
        ifstream infile;
        char fname1[10],fname2[10];
        char ch;
        cout << " enter a file name to be copied ? \n";
        cin >> fname1;
        cout << “ new file name ? \n”;
        cin >> fname2;
        infile.open(fname1);
        if (infile.fail()) {
                cerr << “ No such a file exists \n”;
                exit(1);
        }
        outfile.open(fname2);
        if (outfile.fail()) {
                cerr << “ unable to create a file \n”;
                exit(1);
        }
        while ( !infile.eof()) {
                ch = (char)infile.get();
                outfile.put(ch);
        }
        infile.close();
        outfile.close();
        return 0;
}
```

Output

```
enter a file name to be copied?
data
new le name?
tempdata
```

BINARY FILE OPERATIONS

In C++, by default the file stream operations are performed in text mode but supports binary file operations also. A binary file is a sequential access file in which data are stored and read back one after another in the binary format instead of ASCII characters. For example, a binary file contains integer, floating point number, array of structures, etc. Binary file processing is well suited for the design and development of a complex data base or to read and write a binary information.

In order to open a binary file, it is required to use the following mode:

```
 infile.open(“data”, ios:: binary);
```

PROGRAM 5

A program to open a binary file for storing a set of numbers on a specified file.

```
// storing data on a file
// using binary le operations
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
        ofstream outfile;
        char fname[10];
        float x,y,temp;
        cout << “ enter a file name ? \n”;
        cin >> fname;
        outfile.open(fname,ios::out | ios::binary);
        x = 1.5;
        y = 10.5;
        cout << “x\t temp ” << endl;
        while ( x <= y) {
```

```
                        temp = x*x;
                        outfile << x << '\t' << temp << endl;
                        cout << x << '\t' << temp << endl;
                        x = x+1.5;
            }
            outfile.close();
            return 0;
}
```

Output

```
enter a file name?
data
The content of the file called "data"
1.5     2.25
3       9
4.5     20.25
6       36
7.5     56.25
9       81
10.5    110.25
```

CLASSES AND FILE OPERATIONS

Since C++ is an OOP language, it is reasonable to study how objects can be read and written onto a file. This section presents how objects can be written to and read from the external device, normally a disk. The header file <fstream> must be included for handling the file input and output operations. The mode
of fi le operations such as to read, to write and both to read and write should be defined. The binary file operations required to handle the input and output are carried out using the member functions get() and put() for insertion and extraction operators. The member functions read() and write() are used to read and write a stream of objects from a specified file respectively.

(a) **Reading an Object from a File** The read() member function is used to get data for the stream of object from a specified file. The general syntax of the read() member function is
 infile.read((char *) &obj, sizeof(obj));

(b) **Writing an Object to a File** The write() member function is used to save the stream of objects on a specified file. The general syntax of the write() member function is:
infile.write((char *) &obj, sizeof(obj));

PROGRAM 6

A program to read an array of class object of student_info such as name, age, gender, height and weight from the keyboard and to store them on a specified file using read() and write() member functions. Again, the same file is opened for reading and displaying the contents of the file on the screen.

```
// array of class objects and fole operations
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX = 200;
class student_info {
        protected:
                char name[20];
                int age;
                char gender;
                float height;
                float weight;
         public:
                void getdata();
                void display();
};
```

```
void student_info :: getdata()
{
        cout << " name:";
        cin >> name;
        cout << " age:";
        cin >> age;
        cout << " gender:";
        cin >> gender;
        cout << " Height:";
        cin >> height ;
        cout << " Weight:";
        cin >> weight;
}
void student_info :: display()
{
        cout << name << setw(5) << age << setw(10) << gender << setw(5) << height << setw(5) << weight <<
        endl;
}
int main()
{
        student_info obj[MAX];
        fstream infile;
        char fname[10];
        int i,n;
        cout << " enter a file name to be stored ? \n";
        cin >> fname;
        infile.open(fname, ios:: in | ios::out);
        cout << " How many objects are to be stored ? \n";
        cin >>n;
        // reading from the keyboard
        cout << " Enter the following information \n";
        for (i =0; i <= n-1; ++i) {
                int j = i;
                cout << endl;
                cout << " object = " << j+1 << endl;
                obj[i].getdata();
        }
        // storing onto the file
        infile.open(fname, ios::out);
        cout << " storing onto the file......\n";
        for (i=0; i<=n-1; ++i){
                infile.write ((char*) &obj[i],sizeof(obj[i]));
        }
        infile.close();
        // reading from the file
        infile.open(fname, ios::in);
        cout << " reading from the file......\n";
        for (i=0; i<=n-1; ++i){
                infile.read ((char*) &obj[i],sizeof(obj[i]));
                obj[i].display();
        }
        infile.close();
        return 0;
}
```

RANDOM ACCESS FILE PROCESSING

A sequential access file is very easy to create than a random access file. In the sequential access file, data are stored and retrieved one after another. The file pointer always move from the starting of the file to the end of file.

On the other hand, a random access fi le need not necessarily start from the beginning of the file and move towards end of the file. Random access means moving the file pointer directly to any location in the file instead of moving it sequentially. The random access approach is often used with data base files.

Declaring a Random Access File: The random access file must be opened with the following mode of access

Mode of Access	**Meaning**
ios:: in	in order to read a file
ios:: out	in order to write a file
ios:: ata	in order to append
ios:: binary	binary format

The fstream inherits the following member functions in order to move the file pointer in
and around the data base.

Enumerated Value	**File Position**
ios::beg	from the beginning of the file
ios::cur	from the current file pointer position
ios::end	from the end of the file

The following member functions are used to process a random access file.

(a) seekg() The seekg() member function is used to position file operations for random input operations.

For example, *the following program segment shows the positioning of the file operation for a random access file*

```
#include <fstream>
using namespace std;
int main()
{
 fstream infile;
 -------
 -------
            infile.seekg(40); // goto byte number 40
            infile.seekg(40,ios::beg); // same as the above
            infile.seekg(0,ios::end); // goto the end of file
            infile.seekg(0); // goto start of the file
            infile.seekg(-1, ios::cur); // the file pointer is moved
                                                                    // back end by one byte
}
```

(b) seekp() The seekp() member function is used to position file operations for random output operations.

(c) tellg() The tellg() member function is used to check the current position of the input stream.

(d) tellp() The tellp() member function is used to check the current position of the output stream.

PROGRAM 7

A program to demonstrate how to read a character from a random access file using seekg() method.

```
//reading from the file
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
        fstream infile;
        char fname[20];
        char ch;
        cout << “enter a file name ?\n”;
        cin >> fname;
        infile.open(fname, ios::in | ios :: out);
        infile.seekg(5L,ios::beg);
        infile.get(ch);
        cout << “after 5 characters from beginning = ” << ch;
        cout << ‘\n’;
        infile.seekg(-10L,ios::end);
        infile.get(ch);
        cout << “10 characters before end = ” << ch;
        cout << ‘\n’;
        infile.seekg(0,ios::cur);
        infile.get(ch);
        cout << “current character = ” << ch;
        infile.close();
        return 0;
}
```
Output

The file called “data.txt” consists of the following contents:

```
abcdefghijklmnopqrstuvwxyz
enter a file name?
data.txt
after 5 characters from beginning = f
10 characters before end = r
current character = s
```