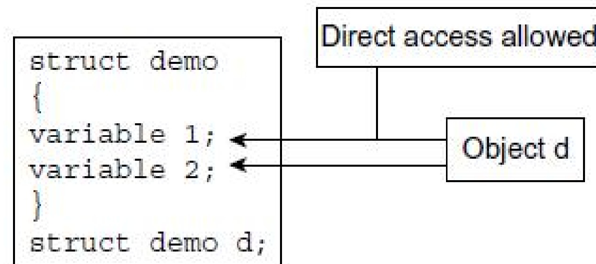
Structure in C:

Structure is a combination of same or different data types. Only variables are declared inside a structure. The initialization of member variables inside the structure is not permitted. Objects can directly access the data members of the structures.

Syntax:	**Example:**

struct <struct name>
{
Variable1;
Variable2;
};
	struct item
	{
	int codeno;
	float prize;
	int qty;
	};



Functions are not permitted as members in structure. Outside functions are also able to access the data members of the structure through the object. Thus, there is no security to data members declared inside the structure as shown in figure

Limitations of structures:

- Functions are allowed as members of structure in C++, but not in C.
- Direct access to data members is possible. So, by default all the structure members are public. Hence, security to data or data hiding is not provided.
- The struct data type is treated as built-in type; that is the use of struct keyword is not necessary to declare objects. But in C it is not possible.
- The data members cannot be initialized inside the structure.

Access to structure members: The data members of a structure are accessed by using object name and operators such as dot (.) or arrow (->). The dot (.) or arrow (->) operators are known as referencing operators. The dot operator is used when simple object is declared and arrow operator is used when object is pointer to structure. The access of members can be accomplished as given in the following syntax:

[Object name][Operator][Member variable name]

When an object is a simple variable, access of members is done as follows:

Object name dot(.) member variable name

Ex. a.codeno
a.price
a.qty

When an object is a pointer to structure then members are accessed as follows:

Ex. a->codeno
a->price
a->qty

Example:

```
#include<iostream>
using namespace std;
struct item // struct declaration
{
        int codeno;
         int qty;
};
int main()
{
        item a,*b; // object declaration
        a.codeno=123; // direct access & initialization of member variables
        a.qty= 150;
        cout<<"\n With simple variable";
        cout<< "\n Codeno :"<<a.codeno;
        cout<<"\n Qty : "<<a.qty;
        b->codeno=124; // direct access & initialization of member variables
        b->qty= 75;
        cout<<"\n With pointer variable";
        cout<< "\n Codeno :"<<a->codeno;
        cout<<"\n Qty : "<<a->qty;
        return 0;
}
```

Classes

A class is an extended concept similar to structures and the class describes both properties (data) and behaviors (functions) of objects i.e. class consists of data members and member functions. Class is not an object but is an instantiate of an object. Class is a blue print for an object which defines the functionality of an object.

Syntax:

```
class <name of class>
{
        private:
                declaration of variables;
                prototype declaration or definition of
                        function;
        public:
                declaration of variables;
                prototype declaration or definition of
                        function;
        protected:
                declaration of variables;
                prototype declaration or definition of
                        function;
};
```

Example:

```
class  Employee
{
        int empno;
        char name[20];
        float sal;
        int deptno;
        public:
                void readData();
                void showData();
};
```

The class is a keyword. The declaration of a class is enclosed with curly braces and terminated with a semicolon. The data members and member functions can be declared in three sections, that is private, protected and public. The private, public and protected keywords are terminated with a colon (**:**).

Declaring Objects:

A class declaration only builds the structure i.e., blue print of an object. The declaration of objects is same as declaration of variables of basic data types. Defining objects of class data type is known as class instantiation. Only when objects are created, memory is allocated to them.

Syntax:

```
name_of_the_class object1, object2,object3;
```

Ex.

```
int x,y,z; // Declaration of integer variables
char a,b,c; // Declaration of character variables
employee a,b, *c; // Declaration of object or class type variables
```

An object is an abstract unit with the following properties:

- It is individual.
- It points to a thing, either physical or logical that is identifiable by the user.
- It holds data as well as operation method that handle data.
- Its scope is limited to the block in which it is defined.

Access to class members: The object can access the public data member and member functions of a class by using dot (.) and arrow (->) operators. The syntax is as follows:

[Object name][Operator][Member name]

To access data members of class the statement would be as follows:

```
Employee e, *a;
e.readData();
```

Where e is an object and readData() is a member function. The dot operator is used because a is a simple object.

In statement

```
a->readData();
```

*a is pointer to class item; therefore, the arrow operator is used to access the member.

Access Specifiers:

1. **Public:** The keyword public can be used to allow objects to access the member variables of a class directly like structures in C.
2. **Private:** The private keyword is used to prevent direct access of member variables by the object. The class by default produces this effect. i.e., a data member can only be accessed by the member functions of the class.
3. **Protected:** The members which are declared under protected section, can be accessed by the member functions of the class and those of the class which is immediately derived from it. No other class can access the protected data elements.

Access Specifiers and their Scope

As described in the table below , the class object can access a public member of the class directly without the use of member function. The private and protected mechanisms do not allow an object to access data directly. The object can access private or protected members only through public member functions of the same class.

Table. Access Limits of class Members

Access Specifiers	Access Permission	
	Class Members	**Outside of the Class Members**
Public	Allowed	Allowed
Private	Allowed	Disallowed
Protected	Allowed	Disallowed

/*C++ program using class **to declare member variable and functions private, public and protected section and make an attempt to access them using object.*/**

```
#include<iostream>
using namespace std;

class sample
{
        private:
                int num;
                void show1()
                {
                cout<<"Inside the private section";
                cout<<"\nEnter a number:";
                cin>>num;
                cout<<"Number is:"<<num;
                }
        public:
                int num1;
                void show()
                {
                        show1();
                        cout<<"\n\nInside the public section";
                        cout<<"\nEnter a number:";
                        cin>>num1;
                        cout<<"Number is:"<<num1;
                        show2();
```

```
            }
    protected:
            int num2;
            void show2()
            {
                    cout<<"\n\nInside the protected section";
                    cout<<"\nEnter a number:";
                    cin>>num2;
                    cout<<"Number is:"<<num2;
            }
};
int main()
{
    sample s;
    s.show();
    return 0;
}
```

Output:

```
Inside the private section
Enter a number:4
Number is:4
Inside the public section
Enter a number:5
Number is:5
Inside the protected section
Enter a number:6
Number is:6
```

Defining Member Functions

In C++ member functions of a class can be defined in two was:

1. Inside the class - The member functions defined inside the class are treated as inline function. If the member function is small then it should be defined inside the class. Otherwise, it should be defined outside the class.
2. Outside the class - If function is defined outside the class, its prototype declaration must be done inside the class.

Member Function inside the Class: Member function inside the class can be declared in public or private section. The following program illustrates the use of a member function inside the class in public section.

/* C++ **program to access private members of a** class **using member function*/**

```
#include<iostream>
using namespace std;

class item
{
	private: // private section starts
		int codeno;
		float price;
		int qty;
	public: // public section starts
		void show() // member function
		{
			codeno=125; // access to private members
			price=195;
			qty=200;
			cout<<"\n Codeno ="<<codeno;
			cout<<"\n Price ="<<price;
			cout<<"\n Quantity="<<qty;
		}
};
int main()
{
	item one; // object declaration
	one.show(); // call to member function
	return 0;
}
```

Output:

```
Codeno =125
Price =195
Quantity=200
```

Private Member Function: In the last section, we learned how to access private data of a class using public member function. It is also possible to declare a function in private section like data variables. To execute private member function, it must be invoked by public member function of

the same class. A member function of a class can invoke any other member function of its own class. This method of invoking function is known as nesting of member function. When one member function invokes other member function, the frequent method of calling function is not used. The member function can be invoked by its name terminated with a semicolon only like normal function. The following program illustrates this point.

/* C++ program to declare private member function and access it using public member function */

```
#include<iostream>
using namespace std;
class item
{
        private: // private section starts
                int codeno;
                float price;
                int qty;
                void values() // private member function
                {
                        codeno=125;
                        price=195;
                        qty=200;
                }
        public: // public section starts
                void show() // public member function
                {
                        values(); // call to private member functions
                        cout<<"\n Codeno ="<<codeno;
                        cout<<"\n Price ="<<price;
                        cout<<"\n Quantity="<<qty;
                }
};
int main()
{
        item one; // object declaration
        // one.values(); // not accessible
        one.show(); // call to public member function
        return 0;
}
```

Output:

```
Codeno =125
Price =195
Quantity=200
```

Member Function Outside the class: To define a function outside the class, following care must be taken.

- The prototype of function must be declared inside the class.
- The function name must be preceded by class name and its return type separated by scope access operator.

Syntax:
return_type class_name::member_function(parameters list)

/* C++ program to define member function of class outside the class.*/

```
#include<iostream>
using namespace std;
class item
{
        private: // private section starts
                int codeno; // member data variables
                float price;
                int qty;
        public: // public section starts
                void show (void); // prototype declaration
}; // end of class
void item:: show() // definition outside the class
{
        codeno=101;
        price=2342;
        qty=122;
        cout<<"\n Codeno ="<<codeno;
        cout<<"\n Price ="<<price;
        cout<<"\n Quantity="<<qty;
}
int main()
{
        item one; // object declaration
        one.show(); // call to public member function
        return 0;
}
```

Output:

```
Codeno =101
Price =2342
Quantity=122
```

Characteristics of Member Functions

1. The difference between member and normal function is that the normal function can be invoked freely, whereas the latter function only by using an object of the same class.
2. The same function can be used in any number of classes. This is possible because the scope of the function is limited to their classes and cannot overlap one another.
3. The private data or private function can be accessed by public member function. Other functions have no access permission.
4. The member function can invoke one another without using any object or dot operator.

Outside Member Function as Inline:

By default, all member functions defined inside the class are inline function. The member function defined outside the class can be made inline by prefixing the keyword inline to function declarator as shown in figure.

inline	return type	class name	::	function name (signature)

/*Write a program to make an outside function as inline.*/

```
class item
{
        private:
                int codeno; // member data variables
                float price;
                int qty;
        public:
                void show (void); // prototype declaration
}; // end of class
inline void item:: show() // outside inline function
{
        codeno=213;
        price=2022;
        qty=150;
        cout<<"\n Code no ="<<codeno;
        cout<<"\n Price ="<<price;
        cout<<"\n Quantity="<<qty;
}
int main()
{
        item one; // object declaration
        one.show(); // call to public member function (inline)
        return 0;
}
```

Output:

```
Code no =213
Price =2022
Quantity=150
```

Rules for Inline Functions:

1. Inline function should be used rarely. It should be applied only at appropriate circumstances.
2. Inline function can be used when the member function contains few statements. For example,

```
inline int item :: square (int x)
{
        return (x*x);
}
```

3. If function takes more time to execute, then it must be declared as inline. The following inline function cannot be expanded as inline.

```
inline void item:: show()
{
        cout<<"\n Codeno ="<<codeno;
        cout<<"\n Price ="<<price;
        cout<<"\n Quantity="<<qty;
}
```

The member function that performs input and output operation requires more times. Inline functions have one drawback, the entire code of the function is placed at the point of call in caller function and it must be noted at compile time. Therefore, inline functions cannot be placed in standard library or run time library.

Data Hiding or Encapsulation

Data hiding is also known as encapsulation. An encapsulated object is often called as an abstract data type. We need to encapsulate data, because programmer often makes various mistakes and the data get changed accidentally. Thus, to protect data, we need to construct a secure and impassable wall to protect the data. Data hiding is nothing but making data variable of the class private. Thus, private data cannot be accessed directly by the object. The objects using public member functions of the same class can access the private data of the class. The keywords private and protected are used for hiding the data.

Classes, Objects, and Memory

Objects are the identifiers declared for class data type. Object is a composition of one more variables declared inside the class. Each object has its own copy of public and private data members. An object can access to its own copy of data members and have no access to data members of other objects.

Only declaration of a class does not allocate memory to the class data members. When an object is declared, memory is reserved for only data member and not for member functions.

/* C++ program to declare objects and display their contents. */

```
#include<iostream>
using namespace std;
class month
{
        public:
                char *name;
                int days;
}; // end of class
int main()
{
        month M1,M3; // object declaration
        M1.name=”January”;
        M1.days=31;
        M3.name=”March”;
        M3.days=31;
        cout<<“\n Object M1 ”;
        cout<<“\n Month name : ”<<M1.name <<“ Address
        :”<<(unsigned)&M1.name;
        cout<<“\n Days :” <<M1.days <<“\t\t Address : ”<<(unsigned)&M1.days;
        cout<<“\n\n Object M3 ”;
        cout<<“\n Month name : ”<<M3.name <<“\t Address :
        ”<<(unsigned)&M3.name;
        cout<<“\n Days :” <<M3.days <<“\t\t Address : ”<<(unsigned)&M3.days;
        return 0;
}
```

Output:

Object M1
Month name : January Address :65522
Days : 31 Address : 65524
Object M3
Month name : March Address : 65518
Days : 31 Address : 65520

Explanation: M1 and M3 are objects of class month. Separate memory is allocated to each object. The contents and address of the member variables are displayed in the output. Figure shows it more visibly.

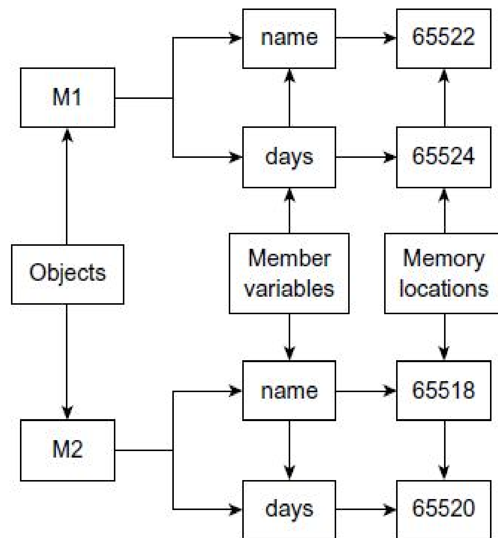
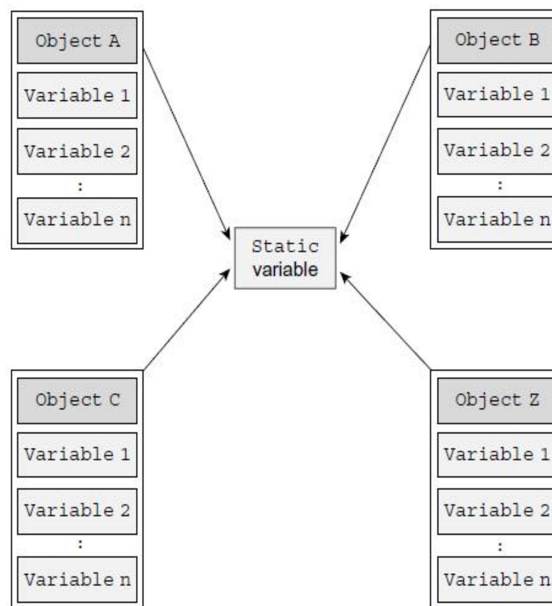


Fig. Memory occupied by objects

static Member Variables

A data member of a class can be qualified as static. A static member variable has certain special characteristics. These are:

- It is initialized to zero when the first object of its class is created.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.
- The static data members are associated with the class and not with any object.
- Static variables are normally used to maintain values common to the entire class.



Syntax:

1. static <variable definition> ;
   Ex.
   ```
   static int c;    // For example in class item
   int item::c=10;
   ```

2. static <function definition>;

/* C++ Program illustrates the use of a static data member.*/

```
#include <iostream>
using namespace std;

class item
{
        static int count;
        int number;
        public:
                void getdata(int a)
                {
                                number = a;
                                count ++;
                }
                void getcount(void)
                {
                        cout << "Count: "<< count <<"\n";
                }
};
int item :: count;

int main()
{
        item a,b,c;
        a.getcount();
        b.getcount();
        c.getcount();

        a.getdata(100);
        b.getdata(200);
        c.getdata(300);

        cout << "After reading data"<<"\n";

        a.getcount();
        b.getcount();
        c.getcount();
        return 0;
}
```

Output:

```
Count: 0
Count: 0
Count: 0
After reading data
Count: 3
Count: 3
Count: 3
```

static Member Functions

Like member variables, function can also be declared as static. When a function is defined as static, it can access only static member variables and functions of the same class. The not-static members are not available to these functions. The static member function declared in public section can be invoked using its class name without using its objects. The programmer must follow the following points while declaring static function:

1. Just one copy of a static member is created in the memory for entire class. All objects of the class share the same copy of static member.
2. static member function can access only static data members or functions.
3. A static member function can be called with the class name as follows:

class-name :: function-name;

class-name :: function-name;

In the following code the static function getcount() displays the number of objects created till that moment.

```
#include <iostream>
using namespace std;

class item
{
        static int count;
        int number;
        public:
        void getdata(int a)
        {
                        number = a;
                        count ++;
        }
        void getcount(void)
        {
                cout << "Count: "<< count <<"\n";
        }
        static void show_count()
        {
                        cout << "Count: "<< count <<"\n";
```

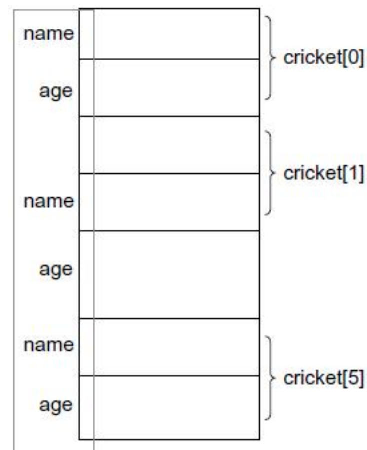
```
        }
};
int item :: count;

int main()
{
        item a,b,c;
        item::show_count();

        a.getdata(100);
        b.getdata(200);
        c.getdata(300);

        cout << "After reading data"<<"\n";

        item::show_count();
        return 0;
}
```

Output:

```
Count: 0
After reading data
Count: 3
```

Array of Objects

Arrays are collection of similar data types. Arrays can be of any data type including user-defined data type created using struct, class, and typedef declarations. We can also create an array of objects. The array elements are stored in continuous memory locations as shown in figure. Consider the following example.




```
class player
{
        private:
                char name [20];
                int age;
        public:
                void input (void);
                void display (void);
};

player cricket[5];
```

/* C ++ program to declare the array of objects. Initialize and display the contents of arrays.*/

```
#include <iostream>
using namespace std;
class player
{
        char name [20];
        int age;
    public:
        void input (void);
        void display (void);
};
void player :: input()
{
    cout<<“\n Enter Palyer name : ”;
    cin>>name;
    cout<<“ Age : ”;
    cin>>age;
}
void player :: display()
{
    cout<<“\n Player name : ”<<name;
    cout<<“\n Age : ”<<age;
}
int main()
{
    player cricket[3]; // array of objects
    cout<<“\n Enter Name and age of 3 players ”;
    for (int i=0;i<3;i++)
    cricket[i].input();
       cout<<“\n The Data Entered by you is ”;
    for (i=0;i<3;i++)
        cricket[i].display();
    return 0;
}
```

Output:

```
Enter Name and age of 3 players
        Enter Palyer name : Sachin  Age : 29
        Enter Palyer name : Rahul   Age : 28
        Enter Palyer name : Saurav Age : 30
The Data Entered by you is
        Player name : Sachin Age : 29
        Player name : Rahul  Age : 28
        Player name : Saurav Age : 30
```

Objects as Function Arguments

The following are the three methods to pass argument to a function:

- Pass-by-value – A copy of object (actual object) is sent to function and assigned to the object of callee function (formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal object are not reflected to actual object.
- Pass-by-reference – Address of object is implicitly sent to function.
- Pass-by-address – Address of the object is explicitly sent to function.

In pass-by-reference and pass-by-address methods, an address of actual object is passed to the function. The formal argument is reference/pointer to the actual object. Hence, changes made in the object are reflected to actual object. These two methods are useful because an address is passed to the function and duplicating of object is prevented.

/*Write a C++ Program to add two complex numbers by passing objects as parameters*/

Example 1. Pass-by-Value

```
#include <iostream>
using namespace std;

class complex
{
        private:
                float real;
                float imag;
        public:
                void read ();
                void show ();
                void sum(complex c1,complex c2);
};
void complex::read ()
{
        cout<<"enter real part";
        cin>>real;

        cout<<"enter imaginary part";
        cin>>imag;
}
```

```
void complex::show ()
{
        cout<<"The Complex Number is:"<<real<<"+i"<<imag<<endl;
}
void complex::sum(complex c1,complex c2)
{
        real=c1.real+c2.real;//here real represents c3's real
        imag=c1.imag+c2.imag;//here imag represents c3's imag
}
int main ()
{
        complex c1,c2,c3;
        c1.read();
        c2.read();

        c3.sum(c1,c2);

        c3.show();
        return 0;

}
```

Output:

```
        enter real part          2
        enter imaginary part:    3
        enter real part:         3
        enter imaginary part:    4
        The Complex Number is:5+i7
```

Example 2. Pass-by-Address

```
#include <iostream>
using namespace std;

class complex
{
                float real;
                float imag;
        public:
                void read ();
                void show ();
                void sum(complex *,complex *);
};
```

```
void complex::read ()
{
        cout<<"enter real part";
         cin>>real;

        cout<<"enter imaginary part";
        cin>>imag;
}
void complex::show ()
{
        cout<<"The Complex Number is:"<<real<<"+i"<<imag<<endl;
}
void complex::sum(complex *c1,complex *c2)
{
        real=c1->real+c2->real;//here real represents c3's real
        imag=c1->imag+c2->imag;//here imag represents c3's imag
}
int main ()
{
        complex c1,c2,c3;
        c1.read();
        c2.read();
        c3.sum(&c1,&c2);
        c3.show();
        return 0;
}
```

Output:

```
enter real part          2
enter imaginary part:    3
enter real part:         3
enter imaginary part:    4
The Complex Number is:5+i7
```

Example 3. Pass-by-Reference

```
class complex
{
        float real;
        float imag;
public:
        void read ();
        void show ();
        void sum(complex &,complex &);
};
```

```
void complex::read ()
{
        cout<<"enter real part";
         cin>>real;
         cout<<"enter imaginary part";
         cin>>imag;
}
void complex::show ()
{
        cout<<"The Complex Number is:"<<real<<"+i"<<imag<<endl;
}
void complex::sum(complex &c1,complex &c2)
{
        real=c1.real+c2.real;//here real represents c3's real
        imag=c1.imag+c2.imag;//here imag represents c3's imag
}
int main ()
{
        complex c1,c2,c3;
        c1.read();
        c2.read();
        c3.sum(c1,c2);
        c3.show();
        return 0;
}
```

Output:

```
enter real part          2
enter imaginary part:  3
enter real part:         3
enter imaginary part:  4
The Complex Number is:5+i7
```

Friend function and Friend class

The main concept of OOP paradigm is 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 an attempt is made directly to access these members, the compiler will display an error message "inaccessible data type". The best way to access private member is to change access specifier to public group. But it violates the concept of **data hiding.** To solve this problem, a friend function/class can be declared to have access these data members. Friend is a special mechanism for letting non-member functions access private data. The keyword **friend** inform the compiler it is not a member of the class.

The general syntax is:

friend return_type function_name(parameters);

Example: friend function

```
class myclass
{
        int num1,num2;
public:
        myclass(int,int);
        void printvalues();
        friend int sum(myclass);
};
myclass::myclass(int n1,int n2)
{
        num1=n1;
        num2=n2;
}
void myclass::printvalues()
{
        cout<<"first:"<<num1<<endl;
        cout<<"second:"<<num2<<endl;
}
```

```
int sum(myclass a)
{
        int n=a.num1+a.num2;
        return n;
}
int main() {
        myclass m(10,20);
        m.printvalues();
        cout<<"Sum of values in
object:"<<sum(m)<<endl;
        return 0;
}
```

Example: friend class

```
#include <iostream>
using namespace std;

class myclass
{
	int num1,num2;
public:
	myclass(int,int);
	void printvalues();
	friend class math;
};
myclass::myclass(int n1,int n2)
{
	num1=n1;
	num2=n2;
}
void myclass::printvalues()
{
	cout<<"first value is:"<<num1<<endl;
	cout<<" second is:"<<num2<<endl;
}
```

```
class math
{
public:
	int sum(myclass);
	int multiply(myclass);
};
int math::sum(myclass obj)
{
	return obj.num1+obj.num2;
}
int math::multiply(myclass obj)
{
	return obj.num1*obj.num2;
}
int main() {
	myclass mc(10,20);
	math m;
	cout<<"The sum of two values in myclass object:"<<m.sum(mc)<<endl;
	cout<<"The multiplication of two values in myclass object:"<<m.multiply(mc)<<endl;

	return 0;
}
```

Constructors:

A constructor is defined to be a special member function, which helps in initializing an object while it is declared.

Characteristics of constructors:

(1) Constructor has the same name as that of the class it belongs.
(2) It should be a public member.
(3) It is invoked implicitly at the time of creation of the objects.
(4) Constructors have neither return value nor void.
(5) The main function of constructor is to initialize objects.
(6) Constructor can have default and can be overloaded.
(7) The constructor without arguments is called as default constructor.

Types of Constructors:

Constructors are classified into three types:

1. Default constructor
2. Parameterized constructor
3. Copy constructor
4. Dynamic Constructor

Default Constructor:

A constructor which does not take any arguments is called the **default constructor.** Suppose A is a class, the default constructor in the class takes the following form:

```
A()
{
        Statements;
}
```

The statements within the body of the function assign the values to the member data of the class. Note that the name of the constructor is same as the class name and no return type is specified not even **void**.

Suresh Yadlapati, M. Tech, (Ph. D), Dept. of IT, PVPSIT.

Example:

```
#include <iostream>
using namespace std;
class construct
{
	int x;
	int y;
public:
	construct();
	void display();
};
construct::construct()
{
	x=10;
	y=20;
}
```

```
void construct::display()
{
	cout<<"The first value is:"<<x<<endl;
	cout<<"The second value is:"<<y<<endl;
}

int main() {

	construct dc;
	dc.display();
	return 0;
}
```

Parameterized Constructor: The constructor that can take arguments are called parametrized constructors.

Example:

```
#include <iostream>
using namespace std;

class construct
{
	int x;
	int y;
public:
	construct(int,int);
	void display();
};
construct::construct(int val1,int val2)
{
	x=val1;
	y=val2;

}
```

```
void construct::display()
{
	cout<<"The first value is:"<<x<<endl;
	cout<<"The second value is:"<<y<<endl;
}

int main() {

	construct pc(10,20),pc1(100,200);
	pc.display();
	pc1.display();
	return 0;
}
```

Multiple Constructors in a class/Constructor Overloading:

When more than one constructor is defined in a class, we say that constructor is overloaded.

```
#include <iostream>
using namespace std;
class construct
{
        int x;
        int y;
public:
        construct();
        construct(int);
        construct(int,int);
        void display();
};
construct::construct()
{
        x=10;y=20;
}
construct::construct(int val1)
{
        x=val1;
        y=20;
}
```

```
construct::construct(int val1,int val2)
{
        x=val1;
        y=val2;

}
void construct::display()
{
        cout<<"The first value is:"<<x<<endl;
        cout<<"The second value
is:"<<y<<endl;
}

int main() {

        construct obj1;
        construct obj2(100);
        construct obj3(100,200);
        //object 1
        obj1.display();
        //object 2
        obj2.display();
        //object 3
        obj3.display();
        return 0;
}
```

Copy Constructor:

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.

```
#include <iostream>
using namespace std;

class construct
{
        int x;
        int y;
public:
        construct(int,int);
        construct(construct &);
        void display();
};
construct::construct(int val1,int val2)
{
        x=val1;
        y=val2;

}
 construct::construct(construct &obj)
{
x=obj.x;
        y=obj.y;

}
void construct::display()
{
        cout<<"The first value is:"<<x<<endl;
        cout<<"The second value
is:"<<y<<endl;
}

int main() {

        construct obj1(10,20);
        construct obj2(obj1);
        obj1.display();
        obj2.display();
        return 0;
}
```

Dynamic Constructor:

Sometimes we may require to dynamically allocate memory for objects during their creation. This is done with the help of dynamic constructors. The dynamic constructors use the **new** operator to allocate memory for objects during runtime. A Constructor which allocates memory and initializes an object during runtime is called a dynamic constructor.

Example:

```
Class Dynamic
{       int *a;
        public:
                Dynamic(int n)
                {
```

```
                a=new int[n];
        }
};
```

Here, the member data **a** of the class Dynamic is a pointer to **int** type. The constructor takes an argument **n** of int type and allocates memory for n integers. The address of the first element in the dynamic array is assigned to the pointer **a.**

Example:

```
#include <iostream>
using namespace std;
class integer
{
    int *a;
    int n;
public:
    integer(int m)
{
                n=m;
                a=new int[m];
}
    void display()
    {
        int i;
        for(i=0;i<n;i++)
            cout<<a[i]<<endl;
    }
    void accept()
    {
        int i;
        for(i=0;i<n;i++)
            cin>>a[i];
    }
};
int main() {
    integer v(5),v1(10);
    cout<<"Enter elements";
    v.accept();
    cout<<"Display elements";
    v.display();
    cout<<"Enter elements";
    v1.accept();
    cout<<"Display elements";
    v1.display();
    return 0;
}
```

Destructors:

A destructor is defined to be a special member function of a class, which helps in releasing the memory occupied by an object when it goes out of scope.

Characteristics:

1. It should be public member
2. The name of the destructor should as that of class name and it is preceded with ~.
3. It cannot take any arguments
4. No return type should be specified including void.
5. **delete** operator is to be used to de-allocate the memory allocated by new operator in the constructor.

Example:

```
#include <iostream>
using namespace std;

int count=0;
class integer
{
        int x;
public:
        integer(int y)
        {
                        count++;
                        cout<<"object"<<count<<"created<<endl";
                        x=y;
        }
        ~integer()
        {
                cout<<"object -"<<count<<" destroyed"<<endl;
                count--;
        }
};
int main() {
        integer a(10),b(20);
        return 0;
}
```

Output: Object 1 created
Object 2 created
Object -2 destroyed
object -1 destroyed

The const Member Functions

The member functions of a class can also be declared as constant using const keyword. The constant functions cannot modify any data in the class. The const keyword is suffixed to the function prototype as well as in function definition. If these functions attempt to change the data, compiler will generate an error message.

/***Write a program to declare** const **member function and attempt any operation within it*/**

```
#include<iostream.h>
#include<conio.h>
class A
{
        int c;
        public :
                void add (int a,int b) const
                {
                        // c=a+b; // invalid statement  a+b;
                        cout<<"a+b = "<<a+b;
                }
};
int main()
{
        A a;
        a .add(5,7);
        return 0;
}
```

Output:

a+b = 12

Explanation: In the above program, the class A is declared with one member variable (c) and one constant member function add(). The add() function is invoked with two integers. The constant member function cannot perform any operation. Hence, the expression c=a+b will generate an error. The expression a+b is valid and cannot alter any value.

The volatile Member Function:

In C++, one can declare a member function with volatile specifies. This step leads to call safely the volatile object. Calling volatile member function with volatile object is safe. This concept is supported with the following programming example.

/***Write a program to call a volatile member function from a volatile object.*/**

```
#include<iostream.h>
#include<conio.h>
class A
{
        int x;
        public:
                void f() volatile // The volatile member function
                {
                        int x=10;
                        cout<<"Value of x:"<<++x;
                }
};
int main()
{
        volatile A c; // The c is a volatile object
        c.f(); // Call a volatile member function safely
        return 0;
}
```

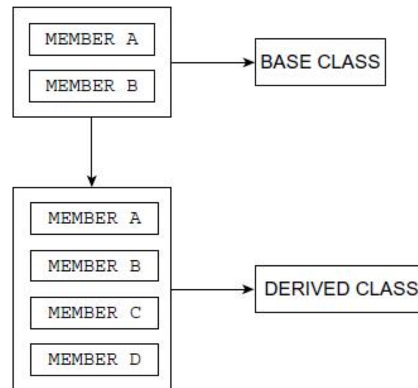
Output:

```
Value of x:11
```

Explanation: The volatile member function f() is called from a volatile object c and the value of x is initialized to 10. Its value is increased by one and displayed on the screen.

Inheritance

Inheritance is one of the most useful and essential characteristics of object-oriented programming. The existing classes are the main components of inheritance. The new classes are created from existing ones. The properties of the existing classes are simply extended to the new classes. The new classes created by using such a method are known as derived classes, and the existing classes are known as base classes, as shown in the figure. The programmer can define new member variables and functions in the derived class. The base class remains unchanged. The object of the derived class can access members of the base as well as derived classes. On the other hand, the object of the base class cannot access members of the derived classes. The base class does not know about their subclasses.



The base class is also called *super class, parent, or ancestor*, and the derived class is called *subclass, child, or descendent*. It is also possible to derive a class from a previously derived class. A class can be derived from more than one class.

Reusability: Reusability means the reuse of properties of the base class in the derived classes. Reusability is achieved using inheritance. Inheritance and reusability are not different from each other. The outcome of inheritance is reusability.

Inheritance Definition: The procedure of creating a new class from one or more existing classes is termed *inheritance*.

Syntax: A new class can be defined as per the syntax given below. The derived class is indicated by associating with the base class. A new class also has its own set of member variables and functions. The syntax given below creates the derived class.

```
class  name_of_the_derived_class: access specifiers  name_of_the_base_class
{
        // member variables of new class (derived class)
}
```

The names of the derived and base classes are separated by a colon (:). The access specifiers may be private, public or protected. The keyword private or public is specified followed by a colon In the absence of an access specifier, the default is private. The access specifiers decide whether the characteristics of the base class are derived privately or publicly. The derived class also has its own set of member variables and functions. The following are the possible syntaxes of declaration:

Ex 1:

```
class B: public A
{
        // Members of class B
};
```

In the above syntax, class A is a base class, and class B is a derived class. Here, the class B is derived publicly.

Ex 2:

```
class B: private A // private derivation
{
        // members of class B
};
```

Ex 3:

```
class B: A // by default private derivation
{
        // members of class B
};
```

Ex 4:

```
class B: protected A // same as private
{
        // members of class B
};
```

It is important to note the following points:

- When a public access specifier is used (Ex. 1), the public members of the base class are public members of the derived class. Similarly, the protected members of the base class are protected members of the derived class.
- When a private access specifier is used, the public and protected members of the base class are the private members of the derived class.

Public Inheritance: When a class is derived publicly, all the public members of the base class can be accessed directly in the derived class.

```
// PUBLIC DERIVATION //
class A      // BASE CLASS
{
        public:
                int x;
};
class B: public A       // DERIVED CLASS
{
        public:
                int y;
};
int main()
{
        B b;            // DECLARATION OF OBJECT
        b.x=20;
        b.y=30;
        cout<<"\n member of A:"<<b.x;
        cout<<"\n Member of B:"<<b.y;
        return 0;
}
```

Output:

```
        Member of A : 20
        Member of B : 30
```

However, in private derivation, an object of the derived class has no permission to directly access even public members of the base class. In such a case, the public members of the base class can be accessed using public member functions of the derived class.

In case the base class has private member variables and a class derived publicly, the derived class can access the member variables of the base class using only member functions of the base class. The public derivation does not allow the derived class to access the private member variable of the class directly as is possible for public member variables. The following example illustrates public inheritance where base class members are declared public and private.

/* Write a program to derive a class publicly from base class. Declare the base class member under private section.*/

```
// PUBLIC DERIVATION //
class A     // BASE CLASS
{
    private:
            int x;
    public:
            A() {x=20;}
            void showx()
            {
                    cout<<"\n x="<<x;
            }
};
class B : public A    // DERIVED CLASS
{
    public:
            int y;
            B() {y=30;}
            void showy()
            {
                    showx();
                    cout<<"\n y="<<y;
            }
};
int main()
{
            B b;      // DECLARATION OF OBJECT
            b.showy();
            return 0;
}
```

<u>Private Inheritance:</u> The objects of the privately derived class cannot access the public members of the base class directly. Hence, the member functions are used to access the members.

/* Write a program to derive a class privately. Declare the member of base class under public section.

```
class A   // BASE CLASS
{
        public:
                int x;
};
```

```
class B : private A    // DERIVED CLASS
{
        public:
                int y;
                B()
                {
                        x=20;
                        y=40;
                }
                void show()
                {
                        cout<<"\n x="<<x;
                        cout<<"\n y="<<y;
                }
};
int main()
{
        B b;    // DECLARATION OF OBJECT
        b.show();
        return 0;
}
```

/*Write a program to derive a class privately.*/

```
class A    // BASE CLASS
{
        int x;
        public:
                A()
                {
                        x=20;
                }
                void showx()
                {
                        cout<<"\n x="<<x;
                }
};
class B : private A // DERIVED CLASS
{
        public:
        int y;
        B()
        {
                 y=40;
        }
```

```
        void showy()
        {
                showx();
                cout<<"\n y="<<y;
        }
};
int main()
{
                B b;    // DECLARATION OF OBJECT
                b.showy();
                return 0;
}
```

Access Specifiers and their Scope:

Sr.No.	Base class access mode	Derived class access mode		
		Private derivation	Public derivation	Protected derivation
A	public	private	public	protected
B	private	Not inherited	Not inherited	Not inherited
C	protected	private	protected	protected

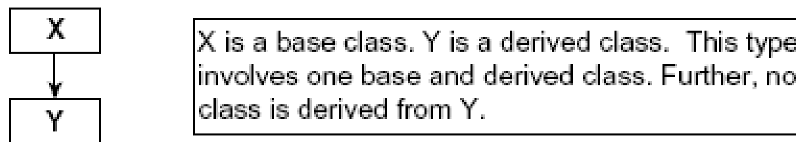
1. All private members of the class are accessible to public members of the same class. They cannot be inherited.
2. The derived class can access the private members of the base class using the member function of the base class.
3. All the protected members of the class are available to its derived classes and can be accessed without the use of the member function of the base class. In other words, we can say that all protected members act as public for the derived class.
4. If any class is prepared for deriving classes, it is advisable to declare all members of the base class as protected, so that derived classes can access the members directly.
5. All the public members of the class are accessible to its derived class. There is no restriction for accessing elements.
6. The access specifier required while deriving class is either private or public. If not specified, private is default for classes and public is default for structures.
7. Constructors and destructors are declared in the public section of the class. If declared in the private section, the object declared will not be initialized and the compiler will flag an error.

Types of Inheritance

Inheritance is classified as follows:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multi-path Inheritance

Single Inheritance: This occurs when only one base class is used for the derivation of a derived class. Further, derived class is not used as a base class, such a type of inheritance that has one base and derived class is known as single inheritance.

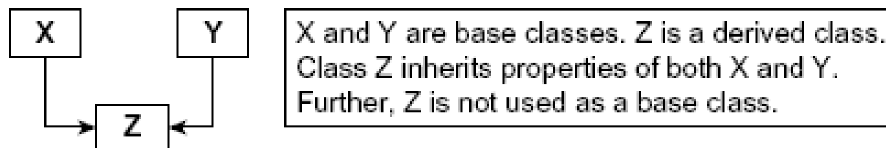


Example:

```
#include <iostream>
using namespace std;

class Publisher
{
        string pname;
        string place;
public:
        void getdata()
        {
                cout<<"Enter name and place of publisher:"<<endl;
                cin>>pname>>place;
        }
        void show()
        {
                cout<<"Publisher Name:"<<pname<<endl;
                cout<<"Place:"<<place<<endl;
        }
};
class Book:public Publisher
{
        string title;
        float price;
```

```
        int pages;
public:
        void getdata()
        {
                Publisher::getdata();
                cout<<"Enter Book Title, Price and No. of pages"<<endl;
                cin>>title>>price>>pages;
        }
        void show()
        {
                Publisher:: show ();
                cout<<"Title:"<<title<<endl;
                cout<<"Price:"<<price<<endl;
                cout<<"No. of Pages:"<<pages<<endl;
        }
};
int main() {

        Book b;
        b.getdata();
        b.show();
        return 0;
}
```

Multiple Inheritance: When two or more base classes are used for the derivation of a class, it is called multiple inheritance.



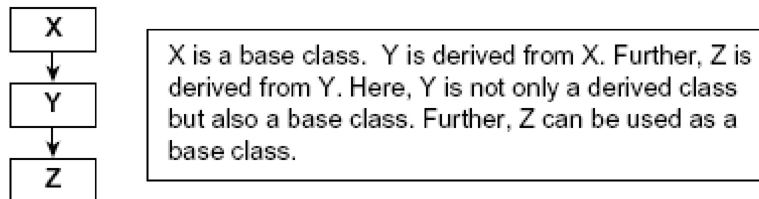
Example:

```
class Publisher
{
        string pname;
        string place;
public:
        void getdata()
        {
                cout<<"Enter name and place of publisher:"<<endl;
                cin>>pname>>place;
        }
        void show ()
```

```
        {
            cout<<"Publisher Name:"<<pname<<endl;
            cout<<"Place:"<<place<<endl;
        }
};
class Author
{
        string aname;
public:
        void getdata()
        {
            cout<<"Enter Author name:"<<endl;
            cin>>aname;
        }
        void show ()
        {
            cout<<"Author Name:"<<aname<<endl;
        }

};
class Book:public Publisher, public Author
{
        string title;
        float price;
        int pages;
public:
        void getdata()
        {
            Publisher::getdata();
            Author::getdata();
            cout<<"Enter Book Title, Price and No. of pages"<<endl;
            cin>>title>>price>>pages;
        }
        void show()
        {
            Publisher:: show ();
            Author:: show ();
            cout<<"Title:"<<title<<endl;
            cout<<"Price:"<<price<<endl;
            cout<<"No. of Pages:"<<pages<<endl;
        }
};
int main() {

        Book b;
        b.getdata();
```

```
        b.show();
        return 0;
}
```

Hierarchical Inheritance: When a single base class is used for the derivation of two or more classes, it is known as hierarchical inheritance.

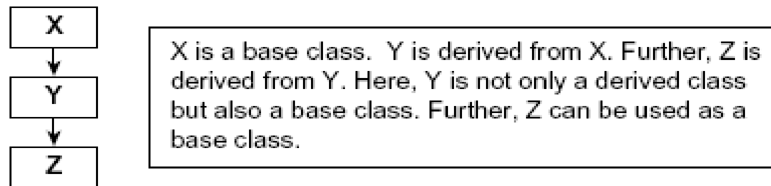


Example:

```
#include <iostream>
using namespace std;

class Account
{
        int act_no;
        string cust_name;
        public:
        void getdata()
        {
                cout<<"Enter Accout number and  Customer name:"<<endl;
                cin>>act_no>>cust_name;
        }
        void show ()
        {
                cout<<"Account Number:"<<act_no<<endl;
                cout<<"Customer Name:"<<cust_name<<endl;
        }
};
class SB_Act: public Account
{
        float roi;
public:
        void getdata()
        {
                Account::getdata();
                cout<<"Enter Rate of Interest"<<endl;
                cin>>roi;
        }
```

```
        void show ()
        {
                cout<<"***** SAVINGS ACCOUNT********"<<endl;
                Account:: show ();
                cout<<"Rate of Interest:"<<roi<<endl;
        }

};
class Current_Act: public Account
{
        float roi;
public:
        void getdata()
        {
                Account::getdata();
                cout<<"Enter Rate of Interest"<<endl;
                cin>>roi;
        }
        void show ()
        {
                cout<<"***** CURRENT ACCOUNT********"<<endl;
                Account:: show ();
                cout<<"Rate of Interest:"<<roi<<endl;
        }

};
int main() {
        /* Saving Account*/
        SB_Act s;
        s.getdata();
        s. show ();
        /* Current Account*/
        Current_Act c;
        c.getdata();
        c. show ();
        return 0;
}
```

Multilevel Inheritance: When a class is derived from another derived class, that is, the derived class acts as a base class, such a type of inheritance is known as multilevel inheritance.

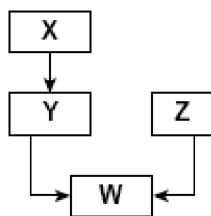


Example:

```
#include <iostream>
using namespace std;

class Publisher
{
        string pname;
        string place;
public:
        void getdata()
        {
                cout<<"Enter name and place of publisher:"<<endl;
                cin>>pname>>place;
        }
        void show ()
        {
                cout<<"Publisher Name:"<<pname<<endl;
                cout<<"Place:"<<place<<endl;
        }
};
class Author:public Publisher
{
        string aname;
public:
        void getdata()
        {
                Publisher::getdata();
                cout<<"Enter Author name:"<<endl;
                cin>>aname;
        }
        void show ()
        {
                Publisher:: show ();
                cout<<"Author Name:"<<aname<<endl;
        }
```

```
};
class Book:public Author
{
        string title;
        float price;
        int pages;
public:
        void getdata()
        {
                Author::getdata();
                cout<<"Enter Book Title, Price and No. of pages"<<endl;
                cin>>title>>price>>pages;
        }
        void show()
        {
                Author:: show ();
                cout<<"Title:"<<title<<endl;
                cout<<"Price:"<<price<<endl;
                cout<<"No. of Pages:"<<pages<<endl;
        }
};
int main() {

        Book b;
        b.getdata();
        b.show();
        return 0;
}
```

Hybrid Inheritance: A combination of one or more types of inheritance is known as hybrid inheritance.



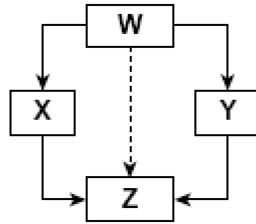
In this type, two types of inheritance is used, i.e. single and multiple inheritance. Class Y is derived from class X. It is single type of inheritance. Further, the derived class Y acts as a base class. The class W is derived from base classes Y and Z. This type of inheritance that uses more than one base class is known as multiple inheritances. Thus, combination of one or more type of inheritance is called as Hybrid inheritance.

Example:

```
#include <iostream>
using namespace std;

class Publisher
{
    string pname;
    string place;
public:
    void getdata()
    {
            cout<<"Enter name and place of publisher:"<<endl;
            cin>>pname>>place;
    }
    void show ()
    {
            cout<<"Publisher Name:"<<pname<<endl;
            cout<<"Place:"<<place<<endl;
    }
};
class Author:public Publisher
{
    string aname;
public:
    void getdata()
    {
            Publisher::getdata();
            cout<<"Enter Author name:"<<endl;
            cin>>aname;
    }
    void show ()
    {
            Publisher:: show ();
            cout<<"Author Name:"<<aname<<endl;
    }

};
class Distributor
{
    string dname;
public:
    void getdata()
    {
            cout<<"Enter Distributor name:"<<endl;
```

```
            cin>>dname;
        }
        void show ()
        {
            cout<<"Distributor Name:"<<dname<<endl;
        }

};
class Book:public Author, public Distributor
{
        string title;
        float price;
        int pages;
public:
        void getdata()
        {
            Author::getdata();
            Distributor::getdata();
            cout<<"Enter Book Title, Price and No. of pages"<<endl;
            cin>>title>>price>>pages;
        }
        void show()
        {
            Author:: show ();
            Distributor:: show ();
            cout<<"Title:"<<title<<endl;
            cout<<"Price:"<<price<<endl;
            cout<<"No. of Pages:"<<pages<<endl;
        }
};
int main() {

        Book b;
        b.getdata();
        b.show();
        return 0;
}
```

Multipath Inheritance: When a class is derived from two or more classes, those are derived from the same base class. Such a type of inheritance is known as multipath inheritance. The multipath inheritance also consists of many types of inheritance, such as multiple, multilevel, and hierarchical, as shown in the figure.



But the disadvantage is ambiguity in classes. Consider the following example:

```
class A1
{
        protected:
                int a1;
};
class A2 : public A1
{
        protected:
                int a2;
};
class A3: public A1
{
        protected:
                int a3;
};
class A4: public A2,A3
{
        int a4;
};
```

In the above example, classes A2 and A3 are derived from class A1; that is, their base class is similar to class A1 (hierarchical inheritance). Both classes A2 and A3 can access the variable a1 of class A1. The class A4 is derived from classes A2 and A3 by multiple inheritance. If we try to access the variable a1 of class A1, the compiler shows error

In the above example, we can observe all types of inheritance, that is, multiple, multilevel, and hierarchical. The derived class A4 has two sets of data members of class A1 through the middle base classes A2 and A3. The class A1 is inherited twice.

Virtual Base Classes: To overcome the ambiguity occurring due to multipath inheritance, the C++ provides the keyword `virtual`. The keyword `virtual` declares the specified classes virtual. The example given below illustrates the virtual classes:

```
class Publisher
{
        string pname;
        string place;
public:
        void getdata()
        {
                cout<<"Enter name and place of publisher:"<<endl;
                cin>>pname>>place;
        }
        void show ()
        {
                cout<<"Publisher Name:"<<pname<<endl;
                cout<<"Place:"<<place<<endl;
        }
};
class Author:virtual public Publisher
{
        string aname;
public:
        void getdata()
        {
                cout<<"Enter Author name:"<<endl;
                cin>>aname;
        }
        void show ()
        {
                cout<<"Author Name:"<<aname<<endl;
        }

};
class Distributor:virtual public Publisher
{
        string dname;
public:
        void getdata()
        {
                cout<<"Enter Distributor name:"<<endl;
                cin>>dname;
        }
```

```
        void show ()
        {
                cout<<"Distributor Name:"<<dname<<endl;
        }

};

class Book:public Author, public Distributor
{
        string title;
        float price;
        int pages;
public:
        void getdata()
        {
                Publisher::getdata();
                Author::getdata();
                Distributor::getdata();
                cout<<"Enter Book Title, Price and No. of pages"<<endl;
                cin>>title>>price>>pages;
        }
        void show()
        {
                Publisher:: show ();
                Author:: show ();
                Distributor:: show ();
                cout<<"Title:"<<title<<endl;
                cout<<"Price:"<<price<<endl;
                cout<<"No. of Pages:"<<pages<<endl;
        }
};
int main() {

        Book b;
        b.getdata();
        b.show();
        return 0;
}
```

Operator Overloading

C++ frequently uses user-defined data types such as classes and structures that are a combination of one or more basic data types. User-defined data types created from class or struct are nothing but a combination of one or more variables of basic data types. The compiler knows how to perform various operations using operators for the built-in types; however, for the objects those are instance of the class, the operation routine must be defined by the programmer.

For example, in traditional programming languages the operators such as +, −, <=, >=, etc. can be used only with basic data types such as `int` or `float`. The operator + (plus) can be used to perform addition of two variables, but the same is not applicable for objects. The compiler cannot perform addition of two objects. The compiler would throw an error if addition of two objects is carried out. The compiler must be made aware of the addition process of two objects. When an expression including operation with objects is encountered, a compiler searches for the definition of the operator, in which a code is written to perform an operation with two objects. Thus, to perform an operation with objects we need to redefine the definition of various operators. For example, for addition of objects A and B, we need to define operator + (plus). Redefining the operator plus does not change its natural meaning. It can be used for both variables of built-in data type and objects of user-defined data type and this is called as ***operator overloading***.

Operator overloading is one of the most valuable concepts introduced by C++ language. It is a type of polymorphism. Polymorphism permits to write multiple definitions for functions and operators.

Operator Overloading Syntax:

Syntax:

```
return_type operator operator_symbol (parameters)
{
        statement1;
        statement2;
}
```

The keyword operator defines a new action or operation to the operator.

Example:

```
Complex operator + (Complex c2)
{
        Complex c3;
        c3.real=real+c2.real;
        c3.imag=imag+c2.imag;
        return c3;
}
```

Operator overloading can be carried out in the following steps.

1. Define a class which is to be used with overloading operations.
2. The public declaration section of the class should contain the prototype of the function operator().
3. Define the definition of the operator() function with proper operations for which it is declared.