Government Arts College(Autonomous)
Coimbatore – 641018

Re-Accredited with 'A' grade by NAAC

Object Oriented Programmimg with C++

Dr. S. Chitra
Associate Professor
Post Graduate & Research Department of Computer Science
Government Arts College(Autonomous)
Coimbatore – 641 018.

Year	Subject Title	Sem.	Sub Code
2018 -19 Onwards	OBJECT ORIENTED PROGRAMMING WITH C++	III	18BCS33C

Objective:

•Learn the fundamentals of input and output using the C++ library
•Design a class that serves as a program module or package.
•Understand and demonstrate the concepts of Functions, Constructor and inheritance.

UNIT – I

Principles of Object Oriented Programming: Software Crisis - Software Evolution - Procedure Oriented Programming - Object Oriented Programming Paradigm - Basic concepts and benefits of OOP - Object Oriented Languages - Structure of C++ Program - Tokens, Keywords, Identifiers, Constants, Basic data type, User-defined Data type, Derived Data type – Symbolic Constants – Declaration of Variables – Dynamic Initialization - Reference Variable – Operators in C++ - Scope resolution operator – Memory management Operators – Manipulators – Type Cast operators – Expressions and their types – Conversions – Operator Precedence - Control Structures

UNIT – II

Functions in C++: Function Prototyping - Call by reference - Return by reference - Inline functions - Default, const arguments - Function Overloading – Classes and Objects - Member functions - Nesting of member functions - Private member functions - Memory Allocation for Objects - Static Data Members - Static Member functions - Array of Objects - Objects as function arguments - Returning objects - friend functions – Const Member functions .

UNIT – III

Constructors: Parameterized Constructors - Multiple Constructors in a class - Constructors with default arguments - Dynamic initialization of objects - Copy and Dynamic Constructors - Destructors - Operator Overloading - Overloading unary and binary operators – Overloading Using Friend functions – manipulation of Strings using Operators.

UNIT – IV

Inheritance: Defining derived classes - Single Inheritance - Making a private member inheritable – Multilevel, Multiple inheritance - Hierarchical inheritance - Hybrid inheritance - Virtual base classes - Abstract classes - Constructors in derived classes - Member classes - Nesting of classes.

UNIT – V

Pointers, Virtual Functions and Polymorphism: Pointer to objects – this pointer- Pointer to derived Class - Virtual functions – Pure Virtual Functions – C++ Streams –Unformatted I/O- Formated Console I/O – Opening and Closing File – File modes - File pointers and their manipulations – Sequential I/O – updating a file :Random access –Error Handling during File operations – Command line Arguments.

TEXT BOOKS

1.E. Balagurusamy, “Object Oriented Programming with C++”, Fourth edition, TMH, 2008.

Unit – III Constructors & Overloading

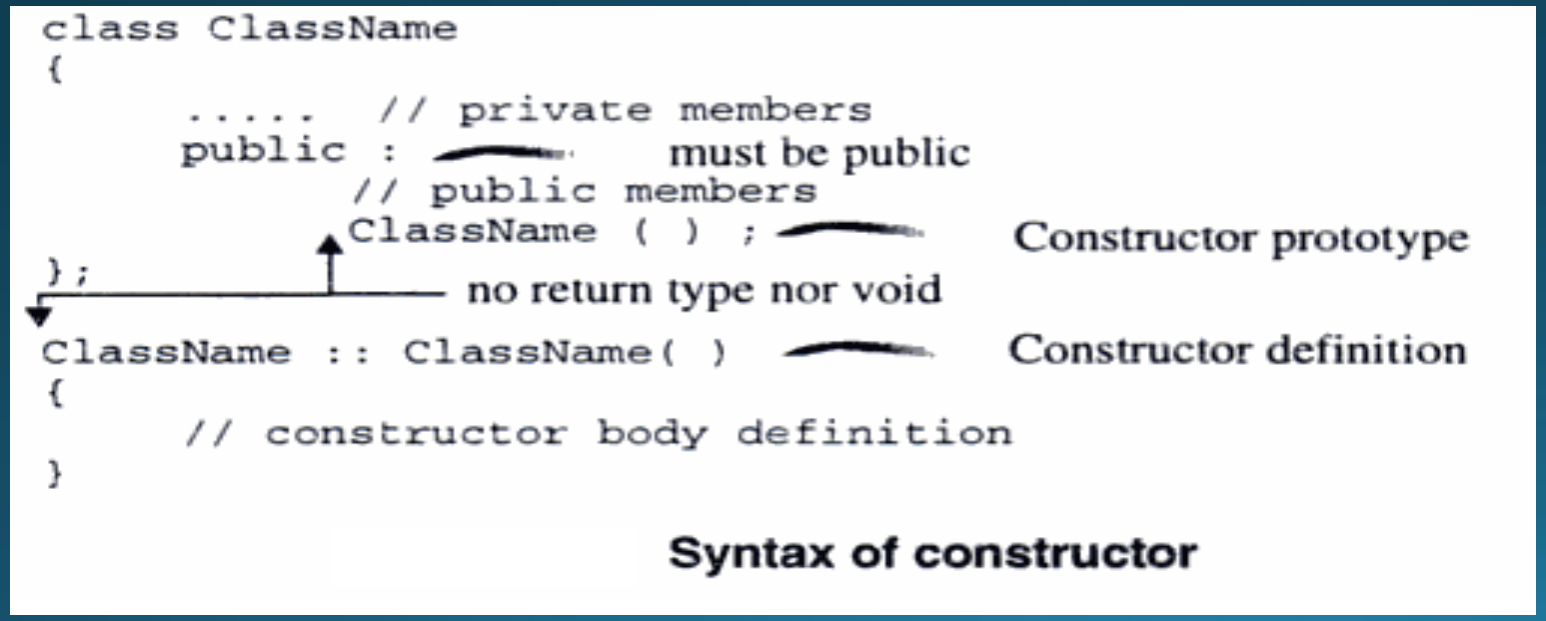
Constructor

*A constructor is a special member function whose task is to initialize the objects of its class.

*It is special because its name is the same as the class name.

*The constructor is invoked when ever an object of its associated class is created.

*It is called constructor because it construct the values of data members of the class.



Syntax of constructor

A constructor is declared and defined as follows:

```
//'class with a constructor
class student
{
int m,n:
public:
student(void);              //constructor declared
------------
------------
};
student :: student(void)
{
m=5;
n=0;
}
```

When a class contains a constructor like the one defined above it is guaranteed that an object created by the class will be initialized automatically.

For example:-

```
student s1;                 //object s1 created
```

This declaration not only creates the object s1 of class integer but also initializes its data members ‘m’ to 5 and ‘n’ to 0.

*A constructor that accept no parameter is called the default constructor.

*The default constructor for class A is A :: A( ).

*If no such constructor is defined, then the compiler supplies a default constructor .

Therefore a statement such as :-

A a1 ; //invokes the default constructor of the class A to create the object "a1" ;

The constructor functions have some characteristics:-

* They should be declared in the public section .

* They are invoked automatically when the objects are created(default constructor).

* They don't have return types, not even void and therefore they cannot return values.

* They cannot be inherited , though a derived class can call the base class constructor .

*Like other C++ function , they can have default arguments,

*Constructor can't be virtual.

* An object with a constructor can't be used as a member of union.

Example of default constructor:

```
#include<iostream.h>
#include<conio.h>
class abc
{
private:
char nm[];
public:
abc( )
{
cout<<”enter your name:\n”;
cin>>nm;
}
void display( )
{
cout<<“\n Name:”<<nm;
getch( );
}
};
int main( )
{
clrscr( );
abc d;
d.display( );
return(0);
}
```

Output:

```
enter your name:
Akil
Name:Akil
```

PARAMETERIZED CONSTRUCTOR:-

*Disadvantage of default constructor is that all objects of the class will have the same initial values. So parameterized constructors were used.
*The constructors that can take arguments are called parameterized constructors.
*Using parameterized constructor, the data members of different objects can be initialized with different values when they are created.

Example:-

```
class integer
constructor defined
{
int m,n;
public:
integer( int x, int y);
--------
---------
};
```

```
integer::integer (int x, int y)    //
{
        m=x;
        n=y;
   }
```

The arguments can be passed to the parameterized constructor by two ways:

1. explicit calling.
2. implicit calling

1. explicit call syntax
classname objectname = classname(arguments);
eg. integer i1 = integer(0,100); // explicit call

2.implicit call syntax
classname objectname(arguments);
integer i2(10,320); //implicit call

CLASS WITH CONSTRUCTOR:-

```
#include<iostream.h>
class integer
{
int m,n;
public:
integer(int,int);
void display(void)
{
cout<<"m=:"<<m<<endl;
cout<<"n="<<n;
}
};
integer :: integer( int x,int y) // constructor defined
{
m=x;
n=y;
}
int main( )
{
integer i1(0, 100); // implicit call
integer i2=integer(25,75);
cout<<" \nobjectl "<<endl;
i1.display( );
cout<<" \n object2 "<<endl;
i2.display( );
}
```

output:
object1
m=0
n=100
object2
m=25
n=75

Example:-

```
#include<iostream.h>
#include<conio.h>
class abc
{
private:
char nm [30];
int age;
public:
abc ( ){ }// default
abc ( char x[], int y);
void get( )
{
cout<<”enter your name:”;
cin>>nm;
cout<<” enter your age:”;
cin>>age;
}
```

```
void display( )
{
cout<<nm«endl;
cout«age;
}
};
abc : : abc(char x[], int y)
{
strcpy(nm,x);
age=y;
}
void main( )
{ abc l;
abc m=abc("computer",20000);
l.get();
l.dispalay( );
m.display ( );
getch( );
}
```

OVERLOADED CONSTRUCTOR:-

```
#include<iostream.h>
#include<conio.h>
class sum
{ private;
int a;
int b;
int c;
float d;
double e;
public:
sum ( )
{
cout<<”enter a;”;
cin>>a;
cout<<”enter b;”;
cin>>b;
cout<<”sum=“<<a+b<<endl;
}
sum(int a,int b);
sum(int a, float d,double c);
};
```

```
sum :: sum(int x,int y)
{
a=x;
b=y; cout<<“\nsum=“<<a+b;
}
sum :: sum(int p, float q ,double r)
{
c=p;
d=q;
e=r; cout<<“\nsum=“<<d+e+c;
}
void main( )
{
clrscr( );
sum l;
sum m=sum(20,50);
sum n= sum(3,3.2,4.55);
getch( );
}
```

output:

```
enter a : 3
enter b : 8
sum=11
sum=70
sum=10.75
```

Dynamic Initialization through Constructors

Object's data members can be dynamically initialized during runtime, even after their creation. The advantage of this feature is that it supports different initialization formats using overloaded constructors. It provides flexibility of using different forms of data at runtime depending upon the user's need.

Consider an example of naming persons. Some persons have only the first name (person name), some have the first and second name (person name and surname), and others have all the three (person name, surname, and third name). The program `name.cpp` illustrates the use of objects for holding names and constructing them at runtime using dynamic initialization.

```
// name.cpp: object with different name pattern
#include <iostream.h>
#include <string.h>
class name
{
    private:
        char first[15];    // first name
        char middle[15];   // middle name
        char last[15];     // last name
    public:
        name()    // constructor0
        {
            // initialize all string pointers to NULL
            first[0] = middle[0] = last[0] = '\0';
        }
        name( char *FirstName );    // constructor1
        name( char *FirstName, char *MiddleName );   // constructor2
          //constructor3
        name( char *FirstName, char *MiddleName, char *LastName );
        void show( char *msg );
};
inline name::name( char *FirstName )
{
    strcpy( first, FirstName );
    middle[0] = last[0] = '\0';    // others to NULL
}
inline name::name( char *FirstName, char *MiddleName )
{
    strcpy( first, FirstName );
    strcpy( middle, MiddleName );
    last[0] = '\0';                    // others to NULL
}
name::name( char *FirstName, char *MiddleName, char *LastName )
{
    strcpy( first, FirstName );
    strcpy( middle, MiddleName );
    strcpy( last, LastName );
}
void name::show( char *msg )
{
    cout << msg << endl;
    cout << "First Name: " << first << endl;
    if( middle[0] )
        cout << "Middle Name: " << middle << endl;
    if( last[0] )
        cout << "Last Name: " << last << endl;
}
void main()
{
    name n1, n2, n3;  // constructor0
    n1 = name( "Rajkumar" );                    // constructor1
    n2 = name( "Savithri", "S" );               // constructor2
    n3 = name( "Venugopal", "K", "R" );         // constructor3
    n1.show( "First person details..." );
    n2.show( "Second person details..." );
    n3.show( "Third person details..." );
}
```

Run

```
First person details...
First Name: Rajkumar
Second person details...
First Name: Savithri
Middle Name: S
Third person details...
First Name: Venugopal
Middle Name: K
Last Name: R
```

The program has four constructors. The arguments to the last three constructors are passed during runtime. The user input is used to initialize the name class's objects in one of the following form:

- No name at all: `default` constructor (constructor0) is invoked
- The first name: constructor1 is invoked
- The first and second name: constructor2 is invoked
- The first, second, and third name: constructor3 is invoked

The compiler selects an appropriate constructor while creating objects by choosing one that matches the input values. For instance, in the situation

```
n2 = name( "Savithri", "S" );          // constructor2
```

the compiler selects the two argument constructor

```
name( char *FirstName, char *MiddleName );  // constructor2
```

which matches the call for initializing the object `n2`'s data members.

Copy Constructor

The parameters of a constructor can be of any of the data types except an object of its own class as a value parameter. Hence declaration of the following class specification leads to an error.

```
class X
{
   private:
    ...
    ...
   public:
      X ( X obj );   // Error: obj is value parameter
       ...
};
```

However, a class's own object can be passed as a reference parameter.

```
class X
{
    ......
    public:
    X()
    X( X &obj );
    X(int a );
};
```

reference to an object of the class X

copy constructor

Copy constructor

Such a constructor having a reference to an instance of its own class as an argument is known as *copy constructor*.

The compiler copies all the members of the user-defined source object to the destination object in the assignment statement, when its members are statically allocated. The data members, which are dynamically allocated must be copied to the destination object explicitly. It can be performed by either using the assignment operator, or the copy constructor.

COPY CONSTRUCTOR:

A copy constructor is used to declare and initialize an object from another object.

Example:-

the statement integer i2(i1);

would define the object i2 and at the same time initialize it to the values of i1.

Another form of this statement is : integer i2=i1;

The process of initialization through a copy constructor is known as copy initialization.

Example:-

```
#include<iostream.h>
class code
{
int id;
public
code ( ) { }                          // defult constructor
code (int a) { id=a; }                // parameterized constructor
code(code &x)                                  // copy constructor
{
id=x.id;
}
```

```
void display( )
{
cout<<id;
}
};
int main( )
{
code A(100);
code B(A);
code C=A;
code D;
D=A;
cout<<" \n id of A :"; A.display( );
cout<<" \nid of B :"; B.display( );
cout<<" \n id of C:"; C.display( );
cout<<" \n id of D:"; D.display( );
}
output :-
id of A:100
id of B:100
id of C:100
id of D:100
```

DYNAMIC CONSTRUCTOR:-

* The constructors can also be used to allocate memory while creating objects.
* This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory.
* Allocation of memory to objects at the time of their construction is known as dynamic constructor of objects.
* The memory is allocated to the object with the help of new operator when the object is created and the memory is deallocated using delete operator when the object is no longer needed.

Example:-

```
#include<iostream.h>
#include<string.h>
class string
{
char *name;
int length;
public:
string ( )
{
length=0;
name= new char [length+1];                    /* one extra for \0 */
}
```

```
string( char *s) //constructor 2
{
length=strlen(s);
name=new char [length+1];
strcpy(name,s);
}
void display(void)
{
cout<<name<<endl;
}
void join(string &a .string &b)
{
length=a. length +b . length;
delete name;
name=new char[length+l]; /* dynamic allocation */
strcpy(name,a.name);
strcat(name,b.name);
}
};
```

```
int main( )
{
char * first = “Joseph” ;
string name1(first),name2(“louis”),naine3( “LaGrange”),sl,s2;
sl.join(name1,name2);
s2.join(s1,name3);
namel.display( );
name2.display( );
name3.display( );
s1.display( );
s2.display( );
}
```

output :-

```
Joseph
Louis
language
Joseph Louis
Joseph Louis Language
```

Constructors with Dynamic Operations

A major application of constructors and destructors is in the management of memory allocation during runtime. It will enable a program to allocate the right amount of memory during execution for each object when the object's data member size is not the same. Allocation of memory to objects at the time of their construction is known as *dynamic construction*. The allocated memory can be released when the object is no longer needed (goes out of scope) at runtime and is known as *dynamic destruction*. The program `vector1.cpp` shows the use of `new` and `delete` operators during object creation and destruction respectively.

```
// vector1.cpp: vector class with array dynamically allocated
#include <iostream.h>
class vector
{
        int *v;   // pointer to a vector
        int sz;   // size of a vector
    public:
        vector( int size )    // constructor
        {
            sz = size;
           v = new int[ size ];     // dynamically allocate vector
        }
        ~vector()             // destructor
        {
            delete v;         // release vector memory
        }
        void read();
        void show_sum();
};
void vector::read()
{
    for( int i = 0; i < sz; i++ )
    {
        cout << "Enter vector[ " << i << " ] ? ";
        cin >> v[i];
    }
}
void vector::show_sum()
{
    int sum = 0;
    for( int i = 0; i < sz; i++ )
        sum += v[i];
    cout << "Vector Sum = " << sum;
}
void main()
{
    int count;
    cout << "How many elements are in the vector: ";
    cin >> count;
    // create an object of vector class and compute sum of vector elements
    vector v1( count );
    v1.read();
    v1.show_sum();
}
```

Run

```
How many elements are in the vector: 5
Enter vector[ 0 ] ? 1
Enter vector[ 1 ] ? 2
Enter vector[ 2 ] ? 3
Enter vector[ 3 ] ? 4
Enter vector[ 4 ] ? 5
Vector Sum = 15
```

DESTRUCTOR:-

* A destructor is used to destroy the objects that have been created by a constructor.
* Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

For Example:-

~ integer( ) { }

* A destructor never takes any argument nor does it return any value.
* It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.
* It is a good practice ses memory space for future use.
* delete is used to free

```
class ClassName
{
      .....  // private members
      public :          must be public
            // public members
            ~ ClassName();          Destructor prototype
};                Tilde character, destructor returns nothing

ClassName :: ~ ClassName()          Destructor definition
{
      // destructor body definition
}
```

Syntax of destructor

Similar to constructors, a destructor must be declared in the `public` section of a class so that it is accessible to all its users. Destructors have no return type. It is incorrect to even declare a `void` return type. *A class cannot have more than one destructor.* The program `test.cpp` illustrates the use of destructors.

```
// test.cpp: a class Test with a constructor and destructor
#include <iostream.h>
class Test
{
   public:                    // 'public' function:
      Test();                 // the constructor
      ~Test();                // the destructor
};
Test::Test()                    // here is the definition of constructor
{
   cout << "constructor of class Test called" << endl;
}
Test::~Test()                   // here is the definition of destructor
{
   cout << "destructor of class Test called" << endl;
}
void main()
{
    Test x;      // constructor is called while creating
    cout << "terminating main()" << endl;
}   // object x goes out of scope, destructor is called
```

Run

```
constructor of class Test called
terminating main()
destructor of class Test called
```

The following rules need to be considered while defining a destructor for a given class:

- The destructor function has the same name as the class but prefixed by a tilde (~). The tilde distinguishes it from a constructor of the same class.
- Unlike the constructor, the destructor does not take any arguments. This is because there is only one way to destroy an object.
- The destructor has neither arguments, nor a return value.
- The destructor has no return type like the constructor, since it is invoked automatically whenever an object goes out of scope.
- There can be only one destructor in each class. This is essentially a violation of the rule that a function can take arguments, thereby making function overloading impossible.

Differences between Constructors and Destructors

The following are the differences between constructors and destructors:

- Arguments cannot be passed to destructors.
- Only one destructor can be declared for a given class as a consequence of the fact that destructors cannot have arguments and hence, destructors cannot be overloaded.
- Destructors can be virtual, while constructors cannot be virtual.

Constructors with Default Arguments

Like other functions in C++, Constructors can also have default arguments

```
// complex1.cpp: default arguments to complex class
#include <iostream.h>
#include <math.h>
class complex
{
   private:
      float real;     // real part of complex number
      float imag;     // imaginary part of complex number
   public:
      complex()       // constructor 0
      {
         real = imag = 0.0;
      }
      complex( float real_in, float imag_in = 0.0 )    // constructor1
      {
         real = real_in;
         imag = imag_in;
      }
      void show( char *msg )      // display complex number in x+iy form
      {
         cout << msg << real;
         if( imag < 0 )
            cout << "-i";
         else
            cout << "+i";
         cout << fabs(imag) << endl;
      }
      complex add( complex c2 ); // Addition of complex numbers
};
// temp = default object + c2;
complex complex::add( complex c2 )      // add default and c2 complex objects
{
    complex temp;                       // object temp of complex class
    temp.real = real + c2.real;         // add real parts
    temp.imag = imag + c2.imag;         // add imaginary parts
    return( temp );                     // return complex object
}
void main()
{
   complex c1( 1.5, 2.0);   // uses constructor1
   complex c2( 2.2 );       // uses constructor1 with default imag value
   complex c3;              // uses constructor0
   c1.show("c1 = ");
   c2.show("c2 = ");
   c3 = c1.add( c2 );       // add c1 and c2 assign to c3
   c3.show( "c3 = c1.add( c2 ): ");
}
```

Run

```
c1 = 1.5+i2
c2 = 2.2+i0
c3 = c1.add( c2 ): 3.7+i2
```

The constructor `complex()`, in the class complex is declared as

```
complex( float real_in, float imag_in = 0.0 )    // constructor1
```

The default value of the argument `imag_in` is zero. Then, the statement in `main()`,

```
complex c2( 2.2 );
```

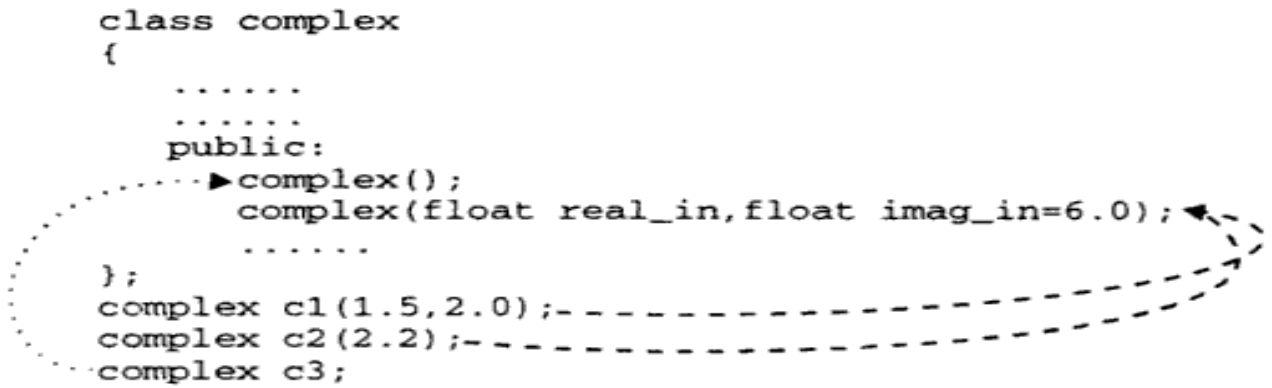
passes only one parameter explicitly to the constructor. The compiler treats this statement as,

```
complex c2( 2.2, 0.0 );
```

by assuming the second argument to have default argument value (`image_in = 0.0`) specified at the declaration of the constructor. However, the statement,

```
complex c1( 1.5, 2.0);
```

assigns 1.5 to `real_in` and 2.0 to `imag_in`. If the actual parameter is explicitly specified, it override the default value. As stated earlier, the missing arguments must be the trailing ones. The invocation o a constructor with default arguments while creating objects of the class `complex` is shown in Figur 11.6.



Default arguments to constructor

Suppose the specification of the constructor `complex(float,float)` is changed to,

```
complex( float real_in = 0.0, float imag_in = 0.0 )
```

in the above program, it causes ambiguity while using a statement such as,

```
complex c1;
```

The confusion is whether to call the no-argument constructor,

```
complex::complex()
```

or the two argument default constructor

```
complex::complex( float = 0.0, float = 0.0)
```

OPERATOR OVERLOADING:-

* Operator overloading is giving additional definition to the C++ opearators.
* All the C++ operators can be overloaded except the following:

1. Class members access operator (. , .*)
2. Scope resolution operator (: :)
3. Size operator(sizeof( ))
4. Conditional operator (? :)

Although the semantics of an operator can be extended, we can't change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associativty.

For example the multiplication operator will enjoy higher precedence than the addition operator.

When an operator is overloaded, its original meaning is not lost.

For example, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

Operator Category	Operators
Arithmetic	`+, -, *, /, %`
Bit-wise	`&, |, ~, ^`
Logical	`&&, ||, !`
Relational	`>, <, ==, !=, <=, >=`
Assignment or Initialization	`=`
Arithmetic Assignment	`+=, -=, *=, /=, %=, &=, |=, ^=`
Shift	`<<, >>, <<=, >>=`
Unary	`++, --`
Subscripting	`[]`
Function Call	`()`
Dereferencing	`->`
Unary Sign Prefix	`+, -`
Allocate and Free	`new, delete`

C++ overloadable operators

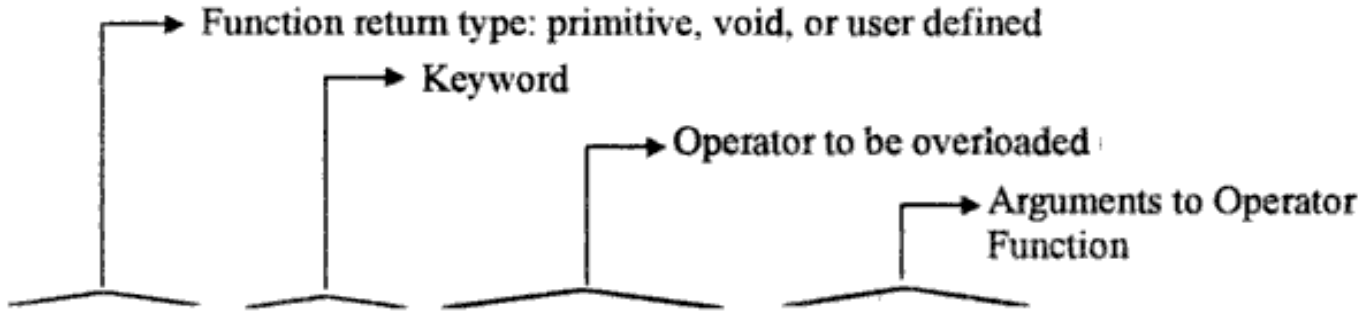
DEFINING OPERATOR OVERLOADING:

- To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied .

- This is done with the help of a special function called operator function, which describes the task.

operator Keyword

The keyword operator facilitates overloading of the C++ operators. The general format of operator overloading is shown The keyword operator indicates that the *operator symbol* following it, is the C++ operator to be overloaded to operate on members of its class. The operator overloaded in a class is known as *overloaded operator function*.



Syntax of operator overloading

- operator functions must be either member function, or friend function.
- A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators.
- This is because the object used to invoke the member function is passed implicitly and therefore is available for the member functions.
- Arguments may be either by value or by reference.

The process of overloading involves the following steps:-

1. Create a class that defines the data type that is used in the overloading operation.

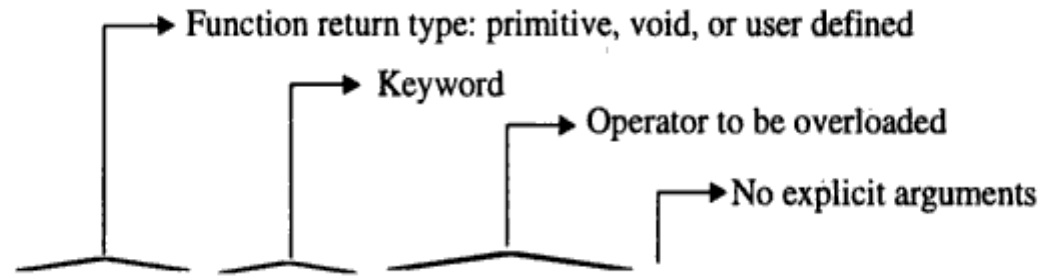
2. Declare the operator function operator op() in the public part of the class

3. It may be either a member function or friend function.

4. Define the operator function to implement the required operations.

Unary Operator Overloading

Overloading without explicit arguments to an operator function is known as *unary operator overloading* and overloading with a single explicit argument is known as *binary operator overloading*. However, with friend functions, unary operators take one explicit argument and binary operators take two explicit arguments. The syntax of overloading the unary operator is shown



Syntax for overloading unary operator

Unary – operator overloading(using member function):

```
class abc
{
int m,n;
public:
abc()
{
m=8;
n=9;
}
void show()
{
cout<<m<<n;
}
void operator -- ()
{
--m;
--n;
}
};
void main()
{
abc x;
x.show();
--x;
x.show();
}
```

Example 2

```
// index2.cpp: Index class with operator overloading
#include <iostream.h>
class Index
{
   private:
      int value;                    // Index Value
   public:
      Index()                       // No argument constructor
      {
         value = 0;
      }
      int GetIndex()                // Index Access
      {
         return value;
      }
      void operator ++()            // prefix or postfix increment operator
      {
         value = value + 1;         // value++;
      }
};
```

```
void main()
{
   Index idx1, idx2;       // idx1 and idx2 are objects of Index class
   // Display index values
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   // Advance Index objects with ++ operators
   ++idx1;              // equivalent to idx1.operator++();
   idx2++;
   idx2++;
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
}
```

Run

```
Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2
```

In `main()`, the statements

```
++idx1;          // equivalent to idx1.operator++();
idx2++;
```

invoke the overloaded ++ operator member function defined in the class `Index`:

```
void operator ++()       // prefix or postfix increment operator
```

The name of this overloaded function is ++. The word `operator` is a keyword and is preceded by the return type `void`. The operator to be overloaded is written immediately after the keyword `operator`. This declarator informs the compiler to invoke the overloaded operator function ++ whenever the unary increment operator is prefixed or postfixed to an object of the `Index` class.

Example 3

```
// index3.cpp: Index class with overloaded operator returning an object
#include <iostream.h>
class Index
{
   private:
      int value;                  // Index Value
   public:
      Index()                     // No argument constructor
      {
         value = 0;
      }
      int GetIndex()               // Index Access
      {
         return value;
      }
      Index operator ++()          // Returns Index object
      {
         Index temp;               // temp object
         value = value + 1;        // update index value
         temp.value = value;       // initialize temp object
         return temp;              // return temp object
      }
};
void main()
{
   Index idx1, idx2;  // idx1 and idx2 are objects of class Index
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   idx1 = idx2++;    //returned object of idx2++ is assigned to idx1
   idx2++;                     // returned object of idx2++ is unused
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
}
```

Run

```
Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2
```

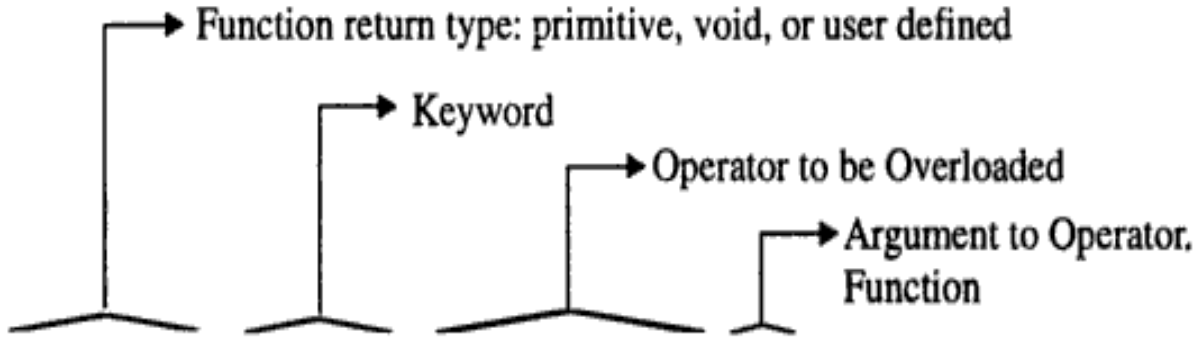
In `main()`, the statement

```
idx1 = idx2++;    //returned object of idx2++ is assigned to idx1
```

invokes the overloaded operator function and assigns the return value to the object `idx1` of the class `Index`. The operator `++()` function creates a new object of the class `Index` called `temp` to be used as a return value; it can be assigned to another object. The `value` data member of the implicit object `idx2` is incremented and then assigned to the `temp` object which is returned to the caller. The returned object is assigned to the destination object `idx1`.

Binary Operator Overloading

The concept of overloading unary operators applies also to the binary operators. The syntax for overloading a binary operator is shown



Syntax for overloading a binary operator

The binary overloaded operator function takes the first object as an implicit operand and the second operand must be passed explicitly. The data members of the first object are accessed without using the dot operator whereas, the second argument members can be accessed using the dot operator if the argument is an object, otherwise it can be accessed directly. Note that, the overloaded binary operator function is a member function defined in the first object's class.

```
// complex2.cpp: Complex Numbers operations with operator overloading
#include <iostream.h>
class complex
{
   private:
      float real;                  // real part of complex number
      float imag;                  // imaginary part of complex number

   public:
      complex()                    // no argument constructor
      {
         real = imag = 0.0;
      }
      void getdata()               // read complex number
      {
         cout << "Real Part ? ";
         cin >> real;
         cout << "Imag Part ? ";
         cin >> imag;
      }
```

```
    complex operator + ( complex c2 );  // complex addition
    void outdata( char *msg )     // display complex number
    {
       cout << endl << msg;
       cout << "(" << real;
       cout << ", " << imag << ")";
    }
};
// add default and c2 complex objects
complex complex::operator + ( complex c2 )
{
   complex temp;                    // object temp of complex class
   temp.real = real + c2.real;      // add real parts
   temp.imag = imag + c2.imag;      // add imaginary parts
   return( temp );                  // return complex object
}
```

```
void main()
{
   complex c1, c2, c3;     // c1, c2, c3 are object of complex class
   cout << "Enter Complex Number c1 .." << endl;
   c1.getdata();
   cout << "Enter Complex Number c2 .." << endl;
   c2.getdata();
   c3 = c1 + c2; // add c1 and c2 and assign the result to c3
   c3.outdata("c3 = c1 + c2: " ); // display result
}
```

Run

```
Enter Complex Number c1 ..
Real Part ? 2.5
Imag Part ? 2.0
Enter Complex Number c2 ..
Real Part ? 3.0
Imag Part ? 1.5
c3 = c1 + c2: (5.5, 3.5)
```

In the class `complex`, the `operator+()` function is declared as follows.

```
complex operator + ( complex c2 );
```

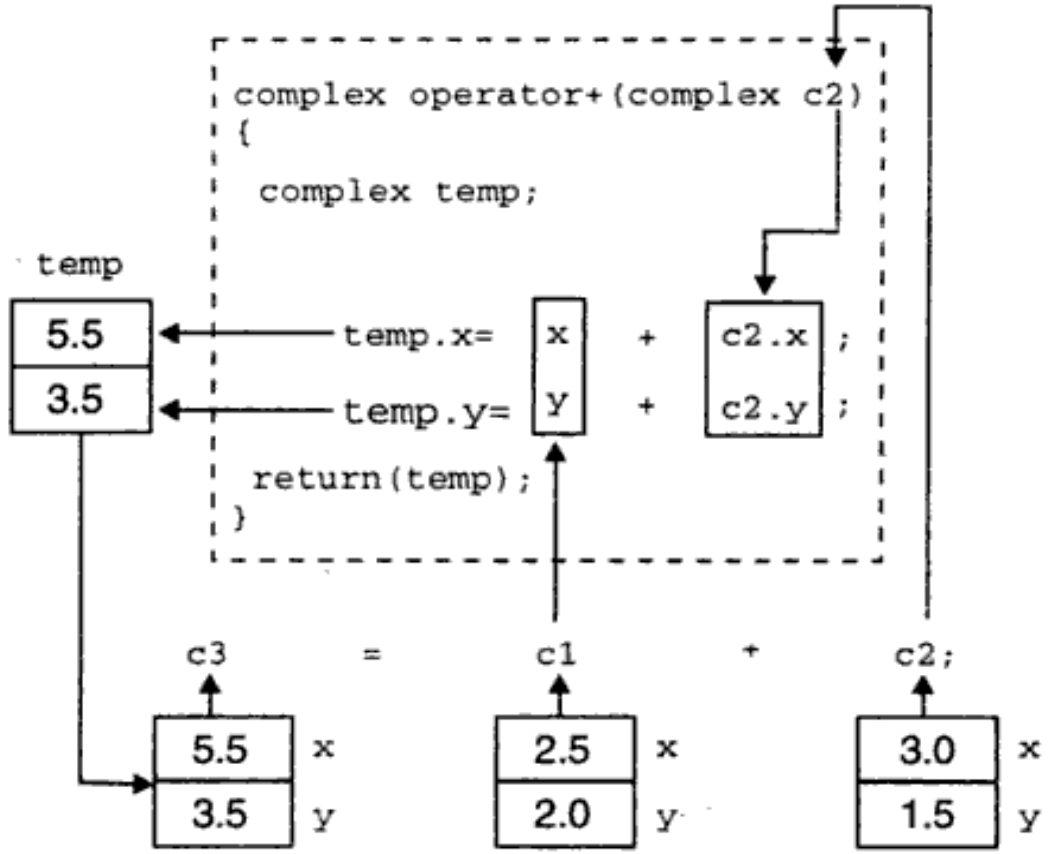
This function takes one explicit argument of type `complex` and returns the result of `complex` type. In a statement such as

```
c3 = c1 + c2;      // c3 = c1.operator+( c2 );
```

it is very important to understand the mechanism of returning a value and relating the arguments of the operator to its objects. When the compiler encounters such expressions, it examines the argument types of the operator. In this case, since the first argument is of type `complex`, the compiler realizes that it must invoke the operator member `+()` function defined in the `complex` class

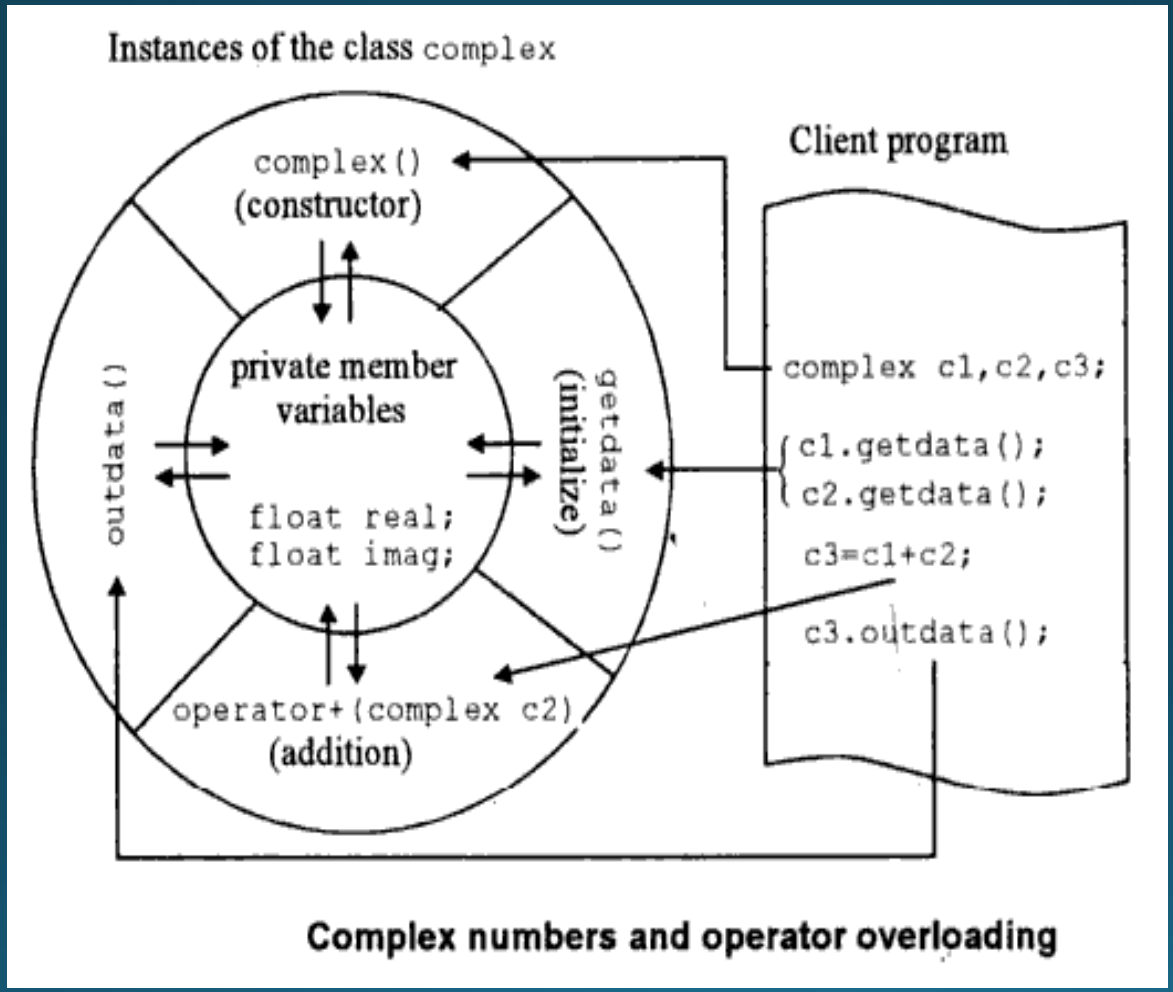
The expression `c1+c2` invokes operator `+()` member function, `c1` object's data members are accessed directly since, this is the object of which the operator function is a member. The right operand is treated as an argument to the function and its members are accessed using the member access dot operator (as `c2.real` and `c2.imag`).

In the overloading of binary operators, as a rule, the *left-hand* operand is used to invoke the operator function and the *right-hand* operand is passed as an argument to the operator function.



Operator overloading in class complex

```
complex complex::operator + ( complex c2 )
{
   complex temp;                    // object temp of complex class
   temp.real = real + c2.real;      // add real parts
   temp.imag = imag + c2.imag;      // add imaginary parts
   return( temp );                  // return complex object
}
```



Complex numbers and operator overloading

Overloading with Friend Functions

Friend functions play a very important role in operator overloading by providing the flexibility denied by the member functions of a class. They allow overloading of stream operators (<< or >>) for stream computation on user defined data types. The only difference between a friend function and member function is that, the friend function requires the arguments to be explicitly passed to the function and processes them explicitly, whereas the member function considers the first argument implicitly. Friend functions can either be used with unary or binary operators.

The prototype of the friend function must be prefixed with the keyword `friend` inside the class body. The body of friend function can appear either inside or outside the body of a class. It is advisable to define a friend function outside the body of a class. The definition of the friend function outside the body of a class is defined as normal function and is not prefixed with the `friend` keyword. The arguments of the friend functions are generally objects of friend classes. In a friend function, all the members of a class (to which this function is a friend) can be accessed by using its objects. *Friena function is not allowed to access members of a class (to which it is a friend) directly, but it can access all the members including the private members by using objects of that class.* Hence, a friend function is similar to a normal function except that it can access the private members of a class using its objects.

Unary Operator Overloading using Friend Functions

The program `complex6.cpp` illustrates the concept of negation of complex numbers. The negation function returns negated object without modifying the source object.

```
// complex6.cpp: Negation of complex number with Unary Operator
#include <iostream.h>
class complex
{
   private:
      float real;
      float imag;
   public:
      complex()          // no argument constructor
      {
         real = imag = 0.0;
       }
      void getdata();    // read complex number
      void outdata( char *msg );     // display complex number
      // overloading of unary minus operator to support c2 = - c1
      friend complex operator - ( complex c1 )
       {
          complex c;
          c.real = -c1.real;
          c.imag = -c1.imag;
          return( c );
       }
      void readdata();
};
```

```
void complex::readdata()
{
   cout << "Real Part ? ";
   cin >> real;
   cout << "Imag Part ? ";
   cin >> imag;
}
void complex::outdata( char *msg )
{
   cout << endl << msg;
   cout << "(" << real;
   cout << ", " << imag << ")";
}
void main()
{
   complex c1, c2;
   cout << "Enter Complex c1.." << endl;
   c1.readdata();
   c2 = -c1;          // invokes complex operator - ()
   c1.outdata( "Complex c1 : " );
   c2.outdata( "Complex c2 = -Complex c1: " );
}
```

Run

```
Enter Complex c1..
Real Part ? 1.5
Imag Part ? -2.5
Complex c1 : (1.5, -2.5)
Complex c2 = -Complex c1: (-1.5, 2.5)
```

The complex number negation function without a friend is declared as follows:

```
complex operator - ()
```

In this case, arguments are implicitly assumed. Using the keyword `friend`, it is declared as follows:

```
friend complex operator - ( complex c1 )
```

The above friend operator function cannot access members of the class `complex` directly, unlike its member functions. In `main()`, the statement

```
c2 = -c1;   // invokes unary operator function, complex operator - ()
```

computes the negation of `c1` and assigns it to `c2`. It returns the negated result without negating contents of the `c1` object. The object `c1` is passed as a value parameter to the negate operator function and any modification to its data members will be reflected in the `c1` object.

The negation operation can also be applied to an object to modify its data members. In this case, the same object acts both as a source and a destination object. It is similar to representing a negative number. This can be achieved by passing the object as a reference parameter to the negation operator function so that, the negation of its data members can be also reflected in the calling object. The program `complex7.cpp` illustrates the concept of negation of complex numbers having the same source and destination operands.

```
// complex7.cpp: Negation of Complex Number with Unary Operator Overloading
#include <iostream.h>
class complex
{
   private:
      float real;
      float imag;
   public:
      complex() { real = imag = 0; }
      void readdata();
      void outdata( char *msg );
      // Note: friend function with explicit reference parameter
      // overloading of unary minus, -c1
      friend void operator - ( complex & c1 ); // definition outside
};
// friend function of the class complex
// Note that, the keyword friend should not prefixed while defining outside
void operator - ( complex & c1 )
{
   c1.real = -c1.real;
   c1.imag = -c1.imag;
}
void complex::readdata()
{
   cout << "Real Part ? ";
   cin >> real;
   cout << "Imag Part ? ";
   cin >> imag;
}
void complex::outdata( char *msg )
{
   cout << endl << msg;
   cout << "(" << real;
   cout << ", " << imag << ")";
}
void main()
{
   complex c1;
   cout << "Enter Complex c1.." << endl;
   c1.readdata();
   -c1;  // invokes unary operator function, complex operator - ()
   c1.outdata( "Result of -Complex c1: " );
}
```

Run

```
Enter Complex c1..
Real Part ? 1.5
Imag Part ? -2.5
Result of -Complex c1: (-1.5, 2.5)
```

In `main()`, the statement

```
-c1;  // invokes unary operator function, complex operator - ()
```

invokes the function

```
void operator - ( complex & c1 )
```

by passing the object `c1` by reference. Thus, the negation of `c1` in the function is also reflected in the calling object. Note that, the definition of operator friend function is the same as normal functions.

Unary – - operator overloading(using friend function):

```
class abc
{
int m,n;
public:
abc()
{
m=8; n=9;
}
void show()
{
cout<<m<<n;
}
friend void operator --(abc p);
};
void operator -- (abc p)
{
--p.m;  --p.n;
}
};
void main()
{
abc x;
x.show();
operator--(x);
x.show();
}
```

Binary Operator Overloading using Friend Function

```
// complex8.cpp: Addition of Complex Numbers with friend feature
#include <iostream.h>
class complex
{
   private:
      float real;
      float imag;
   public:
      complex()
      {}
      complex( int realpart )
      {
         real = realpart;
      }
      void readdata()
      {
         cout << "Real Part ? ";
         cin >> real;
         cout << "Imag Part ? ";
         cin >> imag;
      }
      void outdata( char *msg )      // display complex number
      {
         cout << endl << msg;
         cout << "(" << real;
         cout << ", " << imag << ")";
      }
      friend complex operator + ( complex c1, complex c2 );
};
// note that friend keyword and scope resolution operator are not used
complex operator + ( complex c1, complex c2 )
{
   complex c;
   c.real = c1.real + c2.real;
   c.imag = c1.imag + c2.imag;
   return( c );
}
void main()
{
    complex c1, c2, c3 = 3.0;
    cout << "Enter Complex1 c1..:" << endl;
    c1.readdata();
    cout << "Enter Complex2 c2..:" << endl;
    c2.readdata();
    c3 = c1 + c2;
    c3.outdata( "Result of c3 = c1 + c2: " );
    // 2.0 is considered as real part of complex
    c3 = c1 + 2.0;    // c3 = c1 + complex(2.0)
    c3.outdata( "Result of c3 = c1 + 2.0: " );
    // 3.0 is considered as real part of complex
    c3 = 3.0 + c2;    // c3 = complex( 3.0 ) + c2
    c3.outdata( "Result of c3 = 3.0 + c2: " );
}
```

Run

```
Enter Complex1 c1..:
Real Part ? 1
Imag Part ? 2
Enter Complex2 c2..:
Real Part ? 3
Imag Part ? 4
Result of c3 = c1 + c2: (4, 6)
Result of c3 = c1 + 2.0: (3, 2)
Result of c3 = 3.0 + c2: (6, 4)
```

In `main()`, the statement

```
c3 = c1 + 2.0;    // c3 = c1 + complex(2.0)
```

has an expression, which is a combination of the object `c1` and the primitive floating point constant 2.0. Though, there is no member function matching this expression, the compiler will resolve this by treating the expression as follows:

```
c3 = c1 + complex( 2.0 );
```

The compiler invokes the single argument constructor and converts the primitive value to a new temporary object (here 2.0 is considered as a real part of the complex number) and passes it to the friend operator function:

```
friend complex operator + ( complex c1, complex c2 )
```

The sum of the object `c1` and a new temporary object `complex( 2.0 )` is computed and assigned to object `c3`. The new temporary objects are destroyed immediately after execution of the statement due to which it is created. The above expression can also be written as

```
c3 = 2.0 + c1;
```

Recall that the left-hand operand is responsible for invoking its member function; but this statement has a numeric constant instead of an object. The outcome of either expression is the same, since the compiler treats it as follows:

```
c3 = complex( 2.0 ) + c1;
```

In C++, an object can be used not only to invoke a friend function, but also as an argument to a friend function. Thus, to the friend operator functions, a built-in type operand can be passed either as the first operand or as the second operand.

Binary operator - for subtracting two complex numbers (using friend function)

```
class complex
{
float real,img;
public:
complex()
{
real=0;
img=0;
}
complex(float r,float i)
{
real=r;
img=i;
}
void show()
{
cout<<real<<”-i”<<img;
}
friend complex operator-(complex p,complex q);
};
```

```
complex operator-(complex p,complex q)
{
complex w;
w.real=p.real-q.real;
w.img=p.img-q.img;
return w;
}
};

void main()
{
complex s(3,4);complex t(4,5);
complex m;
m=operator-(s,t);
s.show();t.show();
m.show();
}
```

Overloading an operator does not change its basic meaning. For example assume the + operator can be overloaded to subtract two objects. But the code becomes unreachable.

```
class integer
{
int x, y;
public:
int operator + ( ) ;
}
int integer: : operator + ( )
{
return (x-y) ;
}
```

Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But, those overloaded by means of a friend function take one argument (the object of the relevant class).
Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

Operator to Overload	Arguments passed to the Member Function	Arguments passed to the Friend Function
Unary Operator	**No**	**1**
Binary Operator	**1**	**2**

operator functions are declared in the class using prototypes as follows:-

vector operator + (vector); /./ vector addition

vector operator-( ); //unary minus

friend vector operator + (vector, vector); // vector add

friend vector operator -(vector); // unary minus

vector operator - (vector &a); // substraction

int operator = =(vector); //comparision

friend int operator = =(vector ,vector); // comparision

vector is a data type of class and may represent both magnitude and direction or a series of points called elements.

Manipulation of Strings Using Operators

Concatenation of Strings

Normally, concatenation of strings is performed by using the library function `strcat()` explicitly. To illustrate this concept, consider the strings `str1` and `str2` which are defined as follows:

```
char str1[50] = "Welcome to ";
char str2[25] = "Operator Overloading";
```

The strings `str1` and `str2` are combined, and the result is stored in `str1` by invoking the function `strcat()` as follows:

```
strcat( str1, str2 );
```

On execution `str2` remains unchanged. In C++, such operations can also be performed by defining a string class and overloading the + operator. A statement such as,

```
str1 = str1 + str2;
```

for concatenation of string, (where `str1` and `str2` are the objects of a class `string`) would be perfectly valid. The program `string.cpp` defines a `string` class and uses it to concatenate strings.

```
// string.cpp: Concatenation of strings
#include <iostream.h>
#include <string.h>
const int BUFF_SIZE = 50; // length of string
class string                    // user defined string class
{
   private:
      char str[BUFF_SIZE];
   public:
      string()                  // constructor1 without arguments
      {
         strcpy( str, "" );
      }
      string( char *MyStr )    // constructor2, one argument
      {
         strcpy( str, MyStr );    // MyStr is copied to str
      }
      void echo()     // display string
      {
         cout << str;
      }
      string operator +( string s ) // overloading + operator
      {
         string temp = str;    // creates object and strcpy( temp.str, str );
         strcat( temp.str, s.str ); //  temp.str = temp.str + s.str
         return temp;    // return string object temp
      }
};
void main()
{
   string str1 = "Welcome to ";              // uses constructor2
   string str2 = "Operator Overloading";  // uses constructor2
   string str3;    // uses constructor1, str3.str = NULL
   // display strings of str1, str2, and str3
   cout << "\nBefore str3 = str1 + str2; ..";
   cout << "\nstr1 = ";
   str1.echo();
   cout << "\nstr2 = ";
   str2.echo();
   cout << "\nstr3 = ";
   str3.echo();
   str3 = str1 + str2;  // str1 invokes its operator + function with str2
   // display strings of str1, str2, and str3
   cout << "\nAfter str3 = str1 + str2; ..";
   cout << "\nstr1 = ";
   str1.echo();
   cout << "\nstr2 = ";
   str2.echo();
   cout << "\nstr3 = ";
   str3.echo();
}
```

Run

```
Before str3 = str1 + str2; ..
str1 = Welcome to
str2 = Operator Overloading
str3 =
After str3 = str1 + str2; ..
str1 = Welcome to
str2 = Operator Overloading
str3 = Welcome to Operator Overloading
```

The prototype of the string concatenation operator function

```
string operator +( string s ) // overloading + operator
```

indicates that the + operator takes one argument of type `string` object and returns an object of the same type. The concatenation is performed by creating a temporary `string` object `temp` and initializing it with the first string. The second string is added to first string in the object `temp` using the `strcat()` and finally the resultant temporary string object `temp` is returned. In this case, the length of `str1` plus `str2` should not exceed `BUFF_SIZE`. If it exceeds, then the behavior of the program may be unpredictable. It can be overcome by testing the length of `str1` plus `str2` before concatenating them in the operator `+()` function of the `string` class and then taking appropriate actions.

Comparison Operators

```
// idxcmp.cpp: Index comparison with overloading of < operator
#include <iostream.h>
enum boolean { FALSE, TRUE };
class Index
{
   private:
      int value;                  // Index Value
   public:
      Index()                     // No argument constructor
      {
         value = 0;
      }
      Index( int val )            // Constructor with one argument
      {
         value = val;
      }
      int GetIndex()               // Index Access
      {
         return value;
      }
      boolean operator< ( Index idx )   //compare indexes
      {
         return( value < idx.value ? TRUE : FALSE );
      }
};
void main()
{
   Index idx1 = 5;
   Index idx2 = 10;
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   if( idx1 < idx2 )
      cout << "\nIndex1 is less than Index2";
   else
      cout << "\nIndex1 is not less than Index2";
}
```

Run

```
Index1 = 5
Index2 = 10
Index1 is less than Index2
```

The concept of overloading the comparison operator < in the above program is similar to overloading arithmetic operators. The operator function `< ()` returns `TRUE` or `FALSE` depending on the magnitudes of the Index operands.

Strings Comparison

```
// strcmp.cpp: Comparison of strings
#include <iostream.h>
#include <string.h>
const int BUFF_SIZE = 50;     // length of string
enum boolean { FALSE, TRUE };
class string                   // user defined string class
{
   private:
      char str[BUFF_SIZE];
   public:
      string()                 // constructor without arguments
      {
         strcpy( str, "" );
      }
      void read()              // read string
      {
         cin >> str;
         //  cout << str;
      }
      void echo()     // display string
      {
         cout << str;
      }
      boolean operator < ( string s ) // overloading < operator
      {
         if( strcmp( str, s.str ) < 0 )
            return TRUE;  // str < s.str in lexicographical order
         else
           return FALSE;
      }
      boolean operator > ( string s ) // overloading > operator
      {
         if( strcmp( str, s.str ) > 0 )
            return TRUE;     // str > s.str in lexicographical order
         else
            return FALSE;
      }
      boolean operator == ( char *MyStr ) // overloading == operator
      {
         if( strcmp( str, MyStr ) == 0 )
            return TRUE;    // str and MyStr are same
         else
            return FALSE;
      }
};
```

```
void main()
{
   string str1, str2;  // uses constructor 1
   while( TRUE )
   {
      cout << "\nEnter String1 <'end' to stop>: ";
      str1.read();
      if( str1 == "end" )
          break;
      cout << "Enter String2: ";
      str2.read();
      cout << "Comparison Status: ";
      // display comparison status
      // display format: String1 "comparison status <, >, = " String2
      str1.echo();
      if( str1 < str2 )
          cout << " < ";
      else
          if( str1 > str2 )
              cout << " > ";
          else
              cout << " = ";
      str2.echo();
   }
   cout << "\nBye.!! That's all folks.!";
}
```

Run

```
Enter String1 <'end' to stop>: C
Enter String2: C++
Comparison Status: C < C++
Enter String1 <'end' to stop>: Rajkumar
Enter String2: Bindu
Comparison Status: Rajkumar > Bindu
Enter String1 <'end' to stop>: Rajkumar
Enter String2: Venugopal
Comparison Status: Rajkumar < Venugopal
Enter String1 <'end' to stop>: HELLO
Enter String2: HELLO
Comparison Status: HELLO = HELLO
Enter String1 <'end' to stop>: end
Bye.!! That's all folks.!
```

The overloaded operator functions of the class `string` uses the library function `strcmp()` to compare the two strings. The `strcmp(..)` operates as follows:

- It returns 0 if both the strings are equal
- It returns a negative value if the first string is less than the second one
- It returns a positive value if the first string is greater than the second one

The terms *less than*, *greater than*, or *equal to* are used in lexicographic sense to indicate whether the first string appears before or after the second in the alphabetical order.

The prototype of string comparison function

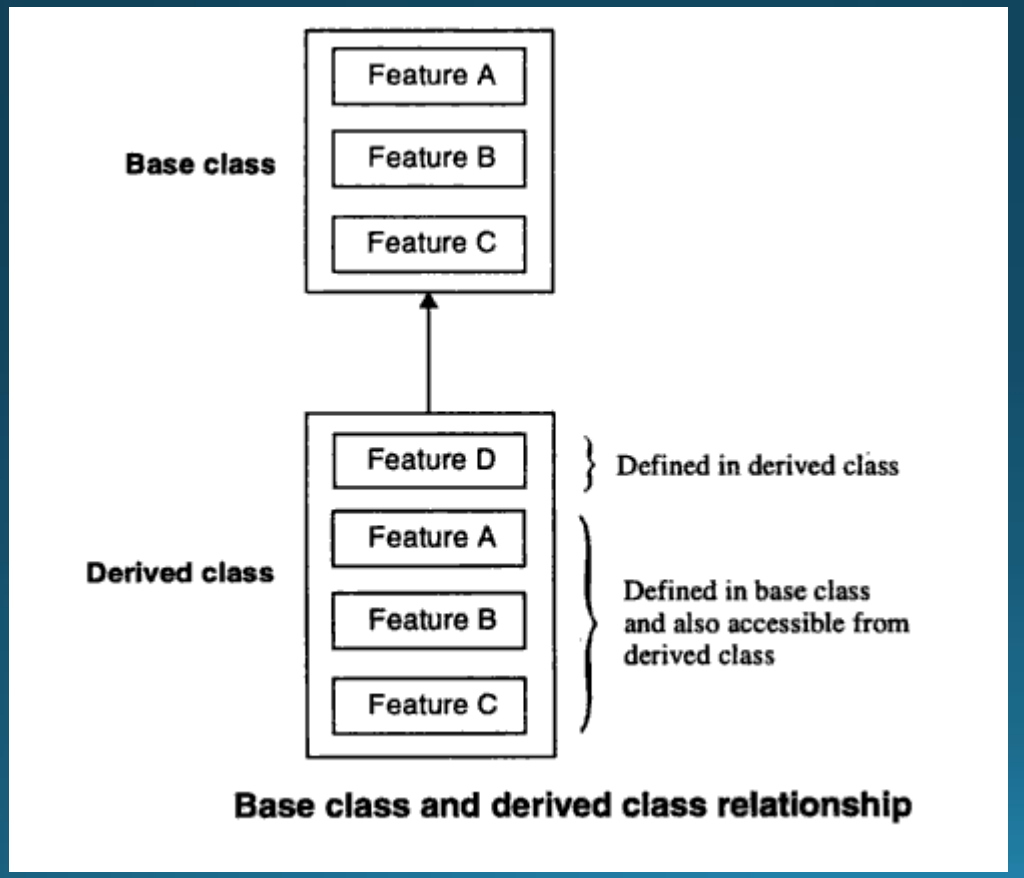
```
boolean operator == ( char *MyStr )
```

indicates that the == operator takes one argument of type pointer to character and returns `TRUE` or `FALSE` depending on the operands weightage in lexicographical order. The `strcmp()` in the function body compares the object's attribute `str` with the argument `MyStr`. From this example, it is understood that the arguments to an overloaded operator need not be of the same data-type, but the overloaded operator must be a *member function of the first object*.

Reusability is yet another feature of OOP. C++ strongly supports the concept of reusability using Inheritance.

Inheritance is a technique of organizing information in a hierarchical form.

Inheritance allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. Classes are created by first inheriting all the variables and behavior defined by some primitive class and then adding specialized variables and behaviors. In object oriented programming, classes encapsulate data and functions into one package. New classes can be built from existing ones. *The technique of building new classes from the existing classes is called inheritance.*



Base class and derived class relationship

Inheritance, a prime feature of OOPs can be stated as *the process of creating new classes (called derived classes), from the existing classes (called base classes)*. The derived class inherits all the capabilities of the base class and can add refinements and extensions of its own. The base class remains unchanged.
derived class inherits the features of the base class (A, B, and C) and adds its own features (D). The arrow in the diagram symbolizes *derived from*. Its direction from the derived class towards the base class, represents that the derived class accesses features of the base class and not vice versa.

```
class ClassName
{
      private:
            ....      // visible to member functions within
            ....      // its class but not in derived class
      protected:
            ....      // visible to member functions within
            ....      // its class and derived class
      public:
            ....      // visible to member functions within
            ....      // its class, derived classes and through object
};
```

```
class X
{
   private:
      int a;
      void f1()
      {
       // .. can refer to members a, b, c, and functions f1, f2, and f3
      }
   protected:
      int b;
      void f2()
      {
       // .. can refer to members a, b, c, and functions f1, f2, and f3
      }
   public:
      int c;
      void f3()
      {
       // .. can refer to members a, b, c, and functions f1, f2, and f3
      }
};
```

The following statements,

```
X objx;                  // objx is an object of class X
int d;          // temporary variable d
```

define the object `objx` of the class `X` and the integer variable `d`. The member access privileges are illustrated by the following statements referring to the object `objx`.

1. Accessing private members of the class X

```
d = objx.a; // Error: 'X::a' is not accessible
objx.f1();  // Error: 'X::f1()' is not accessible
```

Both the statements are invalid because the private members of a class are inaccessible to the object `objx`.

2. Accessing protected members of the class X

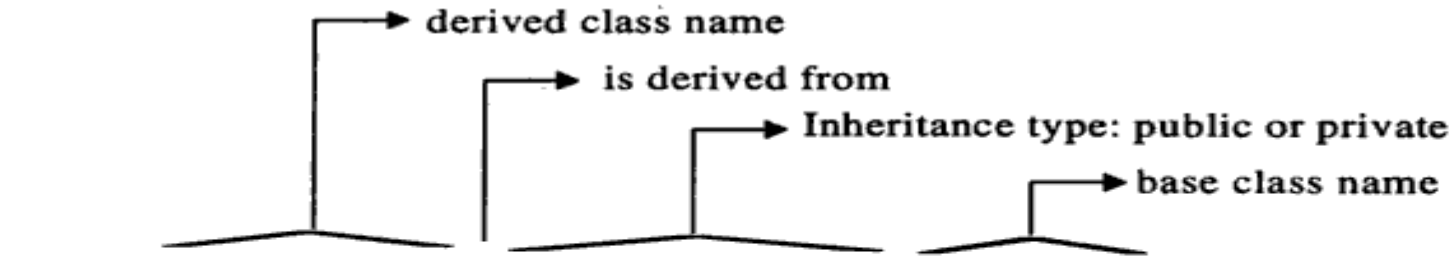
```
d = objx.b; // Error: 'X::b' is not accessible
objx.f2();  // Error: 'X::f2()' is not accessible
```

Both the statements are invalid because the protected members of a class are inaccessible since they are private to the class X.

3. Accessing public members of the class X

```
d = objx.c; // OK
objx.f3();  // OK
```

Both the statements are valid because the public members of a class are accessible to statements outside the scope of the class.



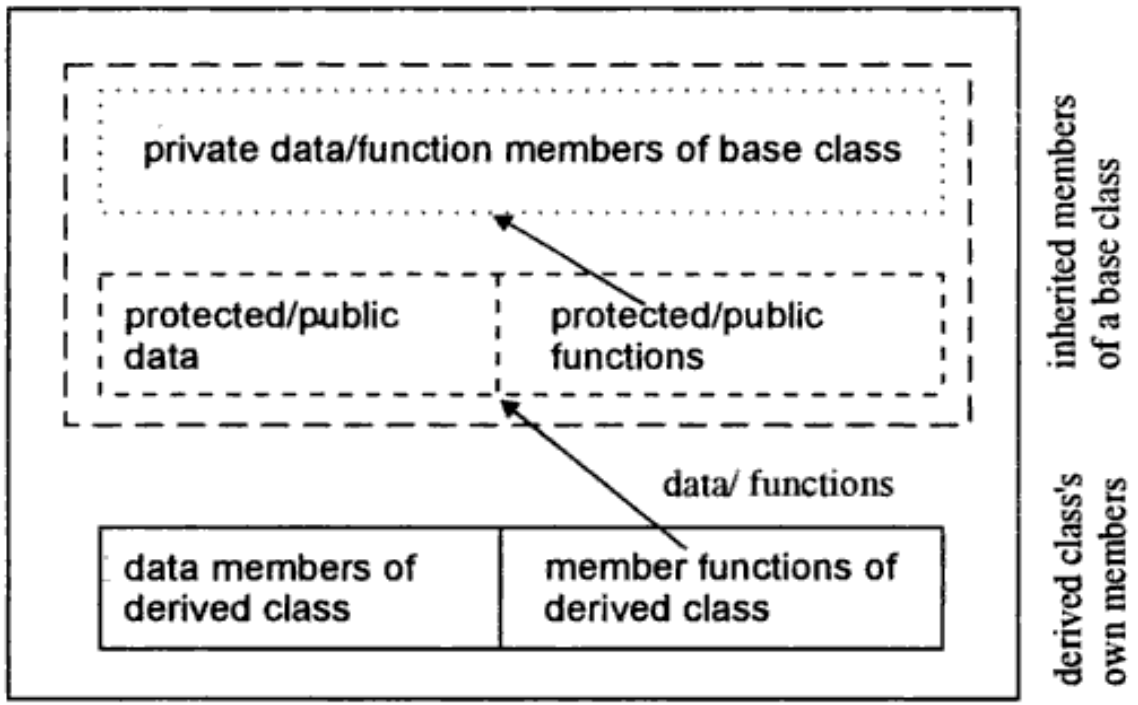

```
class DerivedClass:[VisibilityMode] BaseClass
{
    // members of derived class
    // and they can access members of the base class
};
```

Syntax of derived class declaration

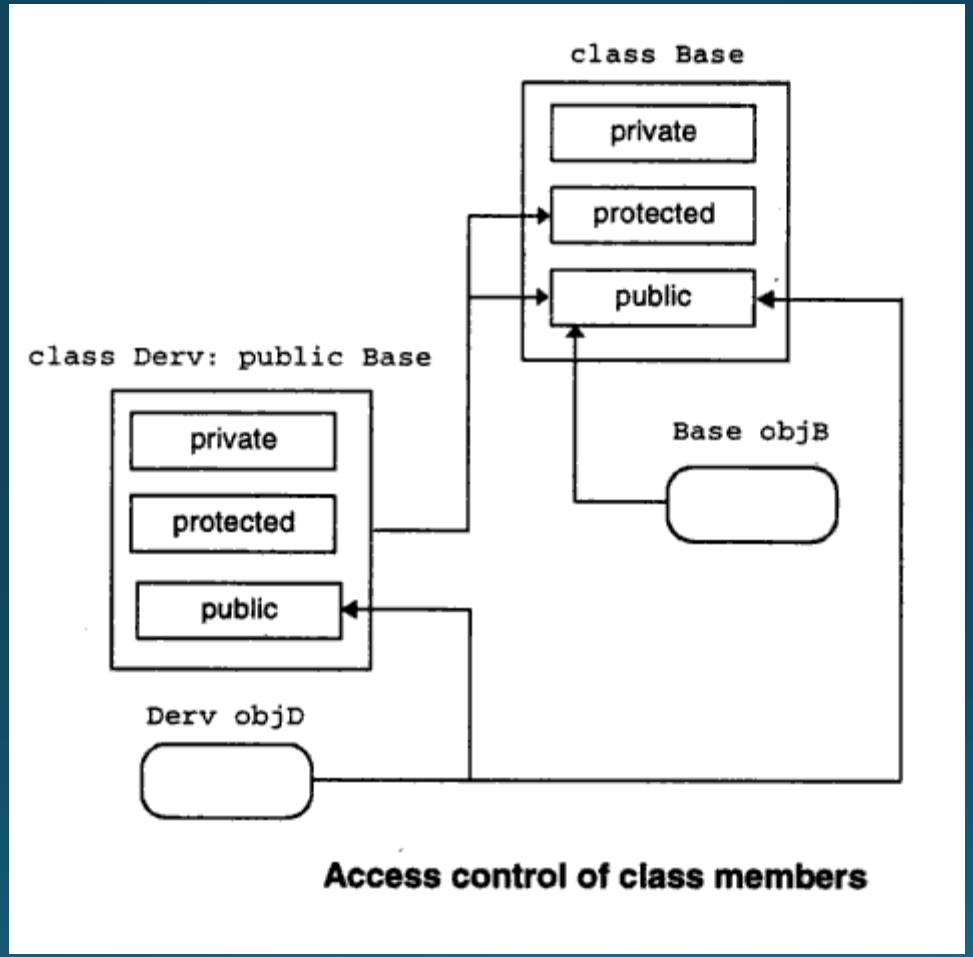
The derivation of `DerivedClass` from the `BaseClass` is indicated by the colon (:). The `VisibilityMode` enclosed within the square brackets implies that it is optional. The default visibility mode is `private`. If the visibility mode is specified, it must be either `public` or `private`. Visibility mode specifies whether the features of the base class are *publicly* or *privately inherited*.

The following are the three possible styles of derivation:

```
1.  class D: public B   // public derivation
    {
        // members of D
    };
2.  class D: private B  // private derivation
    {
        // members of D
    };
3.  class D: B          // private derivation by default
    {
        // members of D
    };
```



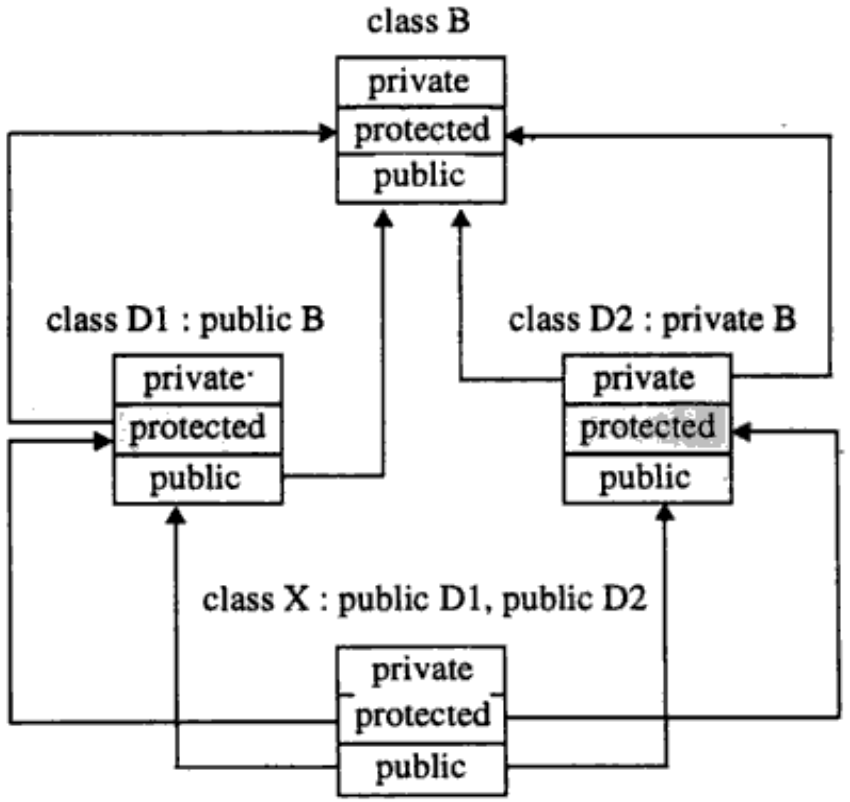
Members of derived class on inheritance



Access control of class members

Function Type	Access directly to		
	Private	Protected	Public
Class Member	Yes	Yes	Yes
Derived class member	No	Yes	Yes
Friend	Yes	Yes	Yes
Friend class member	Yes	Yes	Yes

Access control to class members



Access mechanism in classes

Base class visibility	Derived class visibility	
	Public derivation	Private derivation
`private`	Not Inherited (inherited base class members can access)	Not Inherited (inherited base class members can access)
`protected`	`protected`	`private`
`public`	`public`	`private`

Visibility of class members

Types of Inheritance

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multipath Inheritance

Single Inheritance: Derivation of a class from only one base class is called single inheritance.

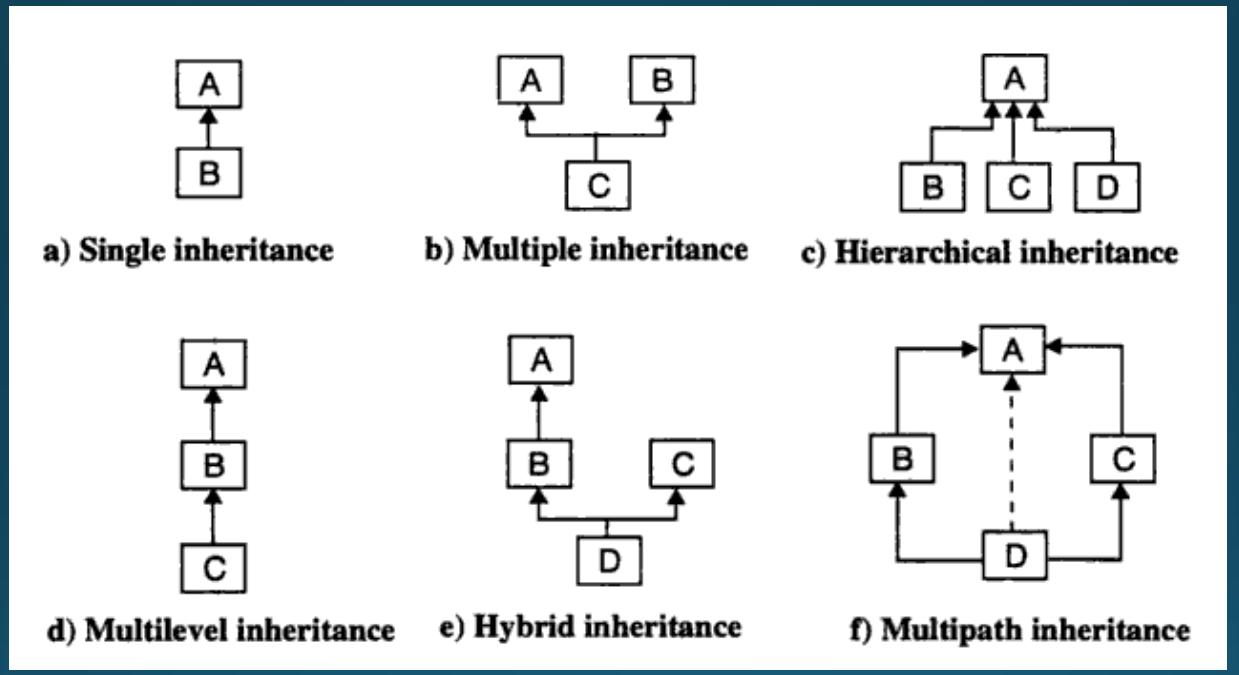
Multiple Inheritance: Derivation of a class from several (two or more) base classes is called multiple inheritance.

Hierarchical Inheritance: Derivation of several classes from a single base class i.e., the traits of one class may be inherited by more than one class, is called hierarchical inheritance.

Multilevel Inheritance: Derivation of a class from another *derived class* is called multilevel inheritance.

Hybrid Inheritance: Derivation of a class involving more than one form of inheritance is known as hybrid inheritance.

Multipath Inheritance: Derivation of a class from other *derived classes*, which are derived from the same base class is called multipath inheritance.


A
B
a) Single inheritance
A
B
C
b) Multiple inheritance
A
B
C
D
c) Hierarchical inheritance
A
B
C
d) Multilevel inheritance
A
B
C
D
e) Hybrid inheritance
A
B
C
D
f) Multipath inheritance

Inheritance and Member Accessibility

1. A private member is accessible only to members of the class in which the private member is declared. They cannot be inherited.
2. A private member of the base class can be accessed in the derived class through the member functions of the base class.
3. A protected member is accessible to members of its own class and to any of the members in a derived class.
4. If a class is expected to be used as a base class in future, then members which might be needed in the derived class should be declared protected rather than private.
5. A public member is accessible to members of its own class, members of the derived class, and outside users of the class.
6. The private, protected, and public sections may appear as many times as needed in a class and in any order. In case an inline member function refers to another member (data or function), that member must be declared before the inline member function is defined. Nevertheless, it is a normal practice to place the private section first, followed by the protected section and finally the public section.
7. The visibility mode in the derivation of a new class can be either private or public.
8. Constructors of the base class and the derived class are automatically invoked when the derived class is instantiated. If a base class has constructors with arguments, then their invocations must be explicitly specified in the derived class's initialization section. However, no-argument constructor need not be invoked explicitly. Remember that, constructors must be defined in the public section of a class (base and derived) otherwise, the compiler generates the error message: *unable to access constructor*.

Constructors in Derived Classes

The constructors play an important role in initializing an object's data members and allocating required resources such as memory. The derived class need not have a constructor as long as the base class has a no-argument constructor. However, if the base class has constructors with arguments (one or more), then it is *mandatory* for the derived class to have a constructor and pass the arguments to the base class constructor. In the application of inheritance, objects of the derived class are usually created instead of the base class. Hence, it makes sense for the derived class to have a constructor and pass arguments to the constructor of the base class. When an object of a derived class is created, the constructor of the base class is executed first and later the constructor of the derived class.

The following examples illustrate the order of invocation of constructors in the base class and the derived class.

1. No-constructors in the base class and derived class

When there are no constructors either in the base or derived classes, the compiler automatically creates objects of classes without any error when the class is instantiated.

```
// cons1.cpp: No-constructors in base class and derived class
#include <iostream.h>
class B      // base class
{
   // body of base class, without constructors
};
class D: public B    // publicly derived class
{
   // body of derived base class, without constructors
   public:
   void msg()
   {
      cout << "No constructors exists in base and derived class" << endl;
   }
};
void main()
{
   D objd; // base constructor
   objd.msg();
}
```

Run

```
No constructors exists in base and derived class
```

2. Constructor only in the base class

```
// cons2.cpp: constructor in base class only
#include <iostream.h>
class B     // base class
{
   public:
   B()
   {
     cout << "No-argument constructor of the base class B is executed";
   }
};
class D: public B    // publicly derived class
{
   public:
};
void main()
{
   D obj1; // accesses base constructor
}
```

Run

```
No-argument constructor of the base class B is executed
```

3. Constructor only in the derived class

```
// cons3.cpp: constructors in derived class only
#include <iostream.h>
class B      // base class
{
   // body of base class, without constructors
};
class D: public B    // publicly derived class
{
   // body of derived base class, without constructors
   public:
   D()
   {
      cout << "Constructos exists in only in derived class" << endl;
   }
};
void main()
{
   D objd;           // accesses derived constructor
}
```

Run

```
Constructos exists in only in derived class
```

4. Constructor in both base and derived classes

```
// cons4.cpp: constructor in base and derived classes
#include <iostream.h>
class B      // base class
{
  public:
  B()
  {
   cout<<"No-argument constructor of the base class B executed first\n";
  }
};
class D: public B    // publicly derived class
{
  public:
   D()
   {
    cout<<"No-argument constructor of the derived class D executed next";
   }
};
void main()
{
   D objd;  // access both base constructor
}
```

Run

```
No-argument constructor of the base class B executed first
No-argument constructor of the derived class D executed next
```

5. Multiple constructors in base class and a single constructor in derived class

```
// cons5.cpp: multiple constructors in base and single in derived classes
#include <iostream.h>
class B     // base class
{
   public:
      B() { cout << "No-argument constructor of the base class B"; }
      B(int a) { cout <<"One-argument constructor of the base class B"; }
};
class D: public B    // publicly derived class
{
   public:
      D( int a )
      { cout << "\nOne-argument constructor of the derived class D"; }
};
void main()
{
   D objd( 3 );
}
```

Run

```
No-argument constructor of the base class B
One-argument constructor of the derived class D
```

6. Constructor in base and derived classes without default constructor

The compiler looks for the no-argument constructor by default in the base class. If there is a constructor in the base class, the following conditions must be met:

- The base class must have a no-argument constructor
- If the base class does not have a default constructor and has an argument constructor, they must be explicitly invoked, otherwise the compiler generates an error.

```
// cons6.cpp: constructor in base and derived class
#include <iostream.h>
class B     // base class
{
   public:
      B(int a) { cout << "One-argument constructor of the base class B";  }
};
class D: public B    // publicly derived class
{
   public:
      D( int a )
      { cout << "\nOne-argument constructor of the derived class D"; }
};
void main()
{
   D objd( 3 );
}
```

The compilation of the above program generates the following error:

```
Cannot find 'default' constructor to initialize base class 'B'
```

This error can be overcome by explicit invocation of a constructor of the base class as illustrated in the program `cons7.cpp`.

7. Explicit invocation in the absence of default constructor

```
// cons7.cpp: constructor in base and derived classes
#include <iostream.h>
class B      // base class
{
   public:
      B(int a)
      { cout << "One-argument constructor of the base class B"; }
};
class D: public B    // publicly derived class
{
   public:
      D( int a ) : B(a)
      { cout << "\nOne-argument constructor of the derived class D"; }
};
void main()
{
   D objd( 3 );
}
```

Run

```
One-argument constructor of the base class B
One-argument constructor of the derived class D
```

In the derived class D, the statement

```
D( int a ):B(a)
```

defines the derived class constructor D(int a) and calls the constructor of the base class using the special form :B(a). Here, the constructor of B is first invoked with an argument a specified in the constructor function D and then the constructor of D is invoked.

8. Constructor in a multiple inherited class with default invocation

```
// cons8.cpp: constructor in base and derived class, order of invocation
#include <iostream.h>
class B1     // base class
{
   public:
      B1() { cout << "\nNo-argument constructor of the base class B1"; }
};
class B2     // base class
{
   public:
      B2() { cout << "\nNo-argument constructor of the base class B2"; }
};
class D: public B2, public B1    // publicly derived class
{
   public:
      D()
      { cout << "\nNo-argument constructor of the derived class D"; }
};
void main()
{
   D objd;
}
```

Run

```
No-argument constructor of the base class B2
No-argument constructor of the base class B1
No-argument constructor of the derived class D
```

The statement

```
class D: public B2, public B1    // publicly derived class
```

specifies that the class `D` is derived from the base classes `B1` and `B2` in order. Hence, constructors are invoked in the order `B2()`, `B1()`, and `D()`; the constructors can be defined with or without arguments.

9. Constructor in a multiple inherited class with explicit invocation

```
// cons9.cpp: constructors with explicit invocation
#include <iostream.h>
class B1    // base class
{
   public:
      B1() {  cout << "\nNo-argument constructor of the base class B1"; }
};
class B2    // base class
{
   public:
      B2() {  cout << "\nNo-argument constructor of the base class B2"; }
};
class D: public B1, public B2
{
   public:
      D(): B2(), B1()    // explicit call to constructors
      {  cout << "\nNo-argument constructor of the derived class D"; }
};
void main()
{
   D objd;
}
```

Run

```
No-argument constructor of the base class B1
No-argument constructor of the base class B2
No-argument constructor of the derived class D
```

In the above program, the statement

```
class D: public B1, public B2    // publicly derived class
```

specifies that, the class `D` is derived from the base classes `B1` and `B2` in order. The statement

```
D(): B2(), B1()
```

in the derived class `D`, specifies that, the base class constructors must be called. However, the constructors are invoked in the order `B1()`, `B2`, and `D`, the order in which the base classes appear in the declaration of the derived class.

10. Constructor in base and derived classes in multiple inheritance

```
// cons10.cpp: constructor in base and derived classes, order of invocation
#include <iostream.h>
class B1    // base class
{
   public:
      B1() { cout << "\nNo-argument constructor of the base class B1"; }
};
class B2    // base class
{
   public:
      B2() { cout << "\nNo-argument constructor of a base class B2"; }
};
class D: public B1, virtual B2      // public B1, private virtual B2
{
   public:
      D(): B1(), B2()
      { cout << "\nNo-argument constructor of the derived class D"; }
};
void main()
{
   D objd;  // base constructor
}
```

Run

```
No-argument constructor of a base class B2
No-argument constructor of the base class B1
No-argument constructor of the derived class D
```

The statement

```
class D: public B1, virtual B2      // public B1, private virtual B2
```

specifies that the class `D` is derived from the base classes `B1` and `B2`. The statement

```
D():B1(), B2()
```

in the derived class `D`, specifies that, the base class constructors must be called. However, the constructors are invoked in the order `B2()`, `B1`, and `D()`, instead of the order in which base classes appear in the declaration of the derived class, since, the virtual base class constructors are invoked first followed by an orderly invocation of constructors of other classes.

11. Constructor in multilevel inheritance

```
// cons11.cpp: constructor in base and derived classes, order of invocation
#include <iostream.h>
class B     // base class
{
   public:
      B() { cout << "\nNo-argument constructor of a base class B"; }
};
class D1: public B     // derived class
{
   public:
      D1() { cout << "\nNo-argument constructor of a base class D1"; }
};
class D2: public D1    // publicly derived class
{
   public:
      D2()
      { cout << "\nNo-argument constructor of a derived class D2"; }
};
void main()
{
   D2 objd;   // base constructor
};
```

Run

```
No-argument constructor of a base class B
No-argument constructor of a base class D1
No-argument constructor of a derived class D2
```

The statement

```
class D2: public D1    // publicly derived class
```

specifies that the class `D2` is derived from the derived class `D1` of `B`. The constructors are invoked in the order `B()`, `D1()`, and `D2()` corresponding to the order of inheritance.

In the derived class, first the constructors of virtual base classes are invoked, second any non-virtual classes, and finally the derived class constructor. Table 14.3 shows the order of invocation of constructors in a derived class.

Method of Inheritance	Order of Execution

`class D: public B`
`{`
`...`
`};`
	B( ): base constructor
	D( ): derived constructor

`class D: public B1, public B2`
`{`
`...`
`};`
	B1( ): base constructor
	B2( ): base constructor
	D( ): derived constructor

`class D: public B1, virtual B2`
`{`
`..`
`};`
	B2( ): virtual base constructor
	B1( ): base constructor
	D( ): derived constructor

`class D1: public B`
`{`
`...`
`};`
`class D2: public D1`
`{`
`..`
`};`
	B( ): super base constructor
	D1( ): base constructor
	D2( ): derived constructor

Order of invocation of constructors

Destructors in Derived Classes

Unlike constructors, destructors in the class hierarchy (parent and child class) are invoked in the reverse order of the constructor invocation. The destructor of that class whose constructor was executed last, while building object of the derived class, will be executed first whenever the object goes out of scope. If destructors are missing in any class in the hierarchy of classes, that class's destructor is not invoked.

```
// cons12.cpp: order of invocation of constructors and destructors
#include <iostream.h>
class B1     // base class
{
   public:
      B1() { cout << "\nNo-argument constructor of the base class B1"; }
       ~B1()
       {
          cout << "\nDestructor in the base class B1";
       }
};
class B2     // base class
{
   public:
      B2() { cout << "\nNo-argument constructor of the base class B2"; }
       ~B2()
       {
          cout << "\nDestructor in the base class B2";
       }
};
class D: public B1, public B2    // publicly derived class
{
   public:
       D()
       { cout << '\nNo-argument constructor of the derived class D"; }
       ~D()
       {
          cout << "\nDestructor in the base class D";
       }
};
void main()
{
   D objd;
}
```

Run

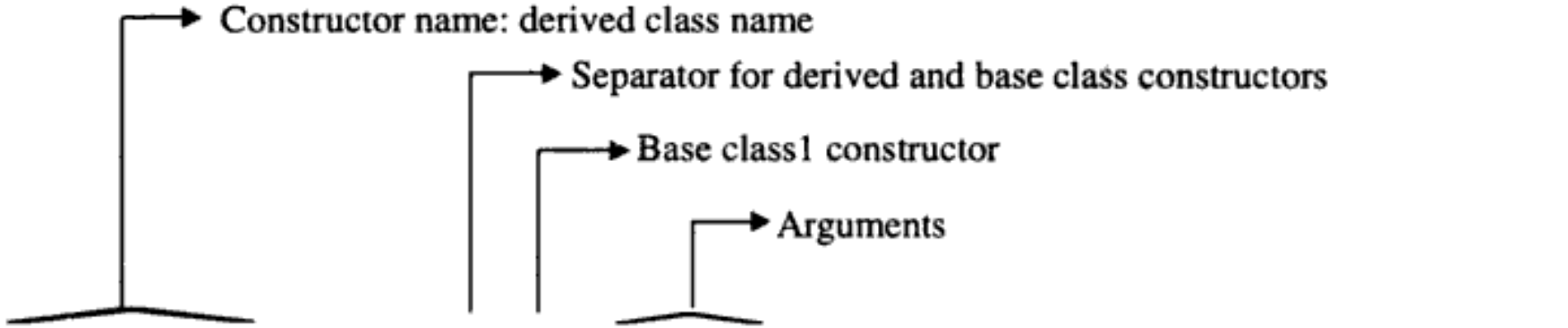
```
No-argument constructor of the base class B1
No-argument constructor of the base class B2
No-argument constructor of the derived class D
Destructor in the base class D
Destructor in the base class B2
Destructor in the base class B1
```

Note that, in this program the constructors are invoked in the order of `B1()`, `B2()`, `D()` whereas, the destructors are invoked in the order of `D()`, `B2()`, `B1()`, which is in reverse order.

In case of dynamically created objects using the `new` operator, they must be destroyed explicitly by invoking the `delete` operator. More specialized class's (which are at the bottom of the hierarchy) destructors are called before a more general one (which are at the top of the hierarchy). As usual, no arguments can be passed to destructors, nor can any return type be declared.

Constructors Invocation and Data Members Initialization

In multiple inheritance, the constructors of base classes are invoked first, *in the order in which they appear in the declaration of the derived class*, whereas in the case of multilevel inheritance, they are executed *in the order of inheritance*. It is the responsibility of the derived class to supply initial values to the base class constructor, when the derived class objects are created. Initial values can be supplied either by the object of a derived class or a constant value can be mentioned in the definition of the constructor. The syntax for defining a constructor in a derived class is shown

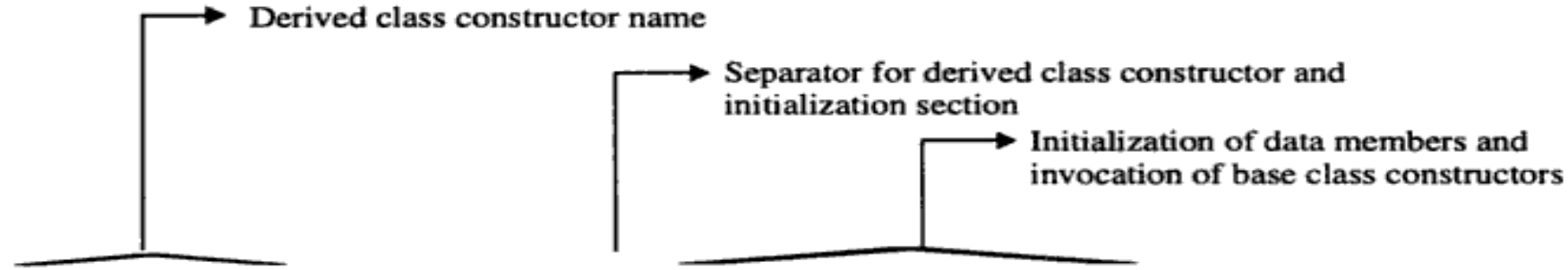



```
DerivedClass(arg_list):Base1(arg_list1),Base2(arg_list2)..,BaseN(arg_listM)
{
   // body of the constructor of derived class
};
```

Syntax of derived class constructor

The parameters arg_list1, arg_list2, .., arg_listM are the list of arguments passed to the constructor or they can be any constant value those match with the arguments of the *constructor list* of base classes.

C++ supports another method of initializing the objects of classes through the use of the *initialization list* in the constructor function. It facilitates the initialization of data members by specifying them in the header section of the constructor. The general form of this method is shown



Syntax of initialization at derived class constructor

Data member initialization is represented by

```
DataMemberName( value )
```

The data members (DataMemberName) to be initialized are followed by the initialization value enclosed in parentheses (resembles a function call). The `value` can be arguments of a constructor, expression or other data members. In the initialization section, any parameter of the argument-list can be used as an initialization value. The data member to be initialized must be a member of its own class. The program `cons14.cpp` illustrates the use of initialization section of the constructor. The following rules must be noted about the initialization and order of invocation of constructors:

- The initialization statements (in the initialization section) are executed in the order of definition of data members in the class.
- Constructors are invoked in the order of inheritance. However, the following rules apply when class is instantiated: first, the constructors of virtual base classes are invoked, second, any non-virtual classes, and finally, the derived class constructor.

```
// cons13.cpp: data members initialization through initialization-section
#include <iostream.h>
class B     // base class
{
   protected:
      int x, y;
   public:
      B(int a, int b): x(a), y(b) {} // x = a, y = b
};
class D: public B    // derived class
{
   private:
      int a, b;
   public:
      D(int p, int q, int r): a(p), B( p, q ), b(r) {}
   void output()
   {
      cout << "x = " << x << endl;
      cout << "y = " << y << endl;
      cout << "a = " << a << endl;
      cout << "b = " << b << endl;
   }
};
void main()
{
   D objb(5, 10, 15);
   objb.output();
}
```

Run

```
x = 5
y = 10
a = 5
b = 15
```

Single Inheritance

When a class inherits from a single base class, it is known as single inheritance. Following program shows the single inheritance using public derivation.

```
#include<iostream.h>
#include<conio.h>
class worker
{
int age;
char name [10];
public:
void getworker ( );
void showworker( );
};
void worker : : getworker( )
{
cout <<"yout name please"
cin >> name;
cout <<"your age please" ;
cin >> age;
}
void worker :: showworker( )
{
cout <<"\n My name is :"<<name<<"\n My age is :"<<age;
}
class manager : public worker //derived class (publicly)
{
int now;
public:
void getnow( ) ;
void shownow( ) ;
};
```

```
void manager : : getnow( )
{
cout << “number of workers under you”;
cin >> now;
}
void manager :: shownow( )
{
cout <<“\n No. of workers under me are: “ << now;
}
main ( )
{
clrscr ( ) ;
manager M1;
M1.getworker( ); M1.getnow( );
M1.showworker( ); M1.shownow( );
}
```

RUN OUTPUT:

```
Your name please
Ravinder
Your age please
27
number of workers under you
30
```

Then the output will be as follows:

```
My name is : Ravinder
My age is : 27
No. of workers under me are : 30
```

The following program shows the single inheritance by private derivation.

```
#include<iostream.h>
#include<conio.h>
class worker                                    //Base class declaration
{
int age;
char name [10] ;
public:
void getworker( ) ;
void showworker( ) ;
};
void worker : : getworker( )
{
cout << “your name please” ;
cin >> name;
cout << “your age please”;
cin >>age;
}
void worker : showworker( )
{
cout << “\n my name is: “ <<name<< “\n” << “my age is : “ <<age;
}
class manager : private worker                  //Derived class (privately by default)
{
int now;
public:
void getnow( ) ;
void shownow( ) ;
};
```

```
void manager : : getnow( )
{
getworker( );        //calling the get function of base
cout << “number of worker under you”;
cin >>now;
}
void manager : : shownow( )
{
showworker( ) ;
cout << “in no. of worker under me are : “ <<now;
}
main ( )
{
clrscr ( ) ;
manager ml;
ml.getnow( ) ;
ml.shownow( );
}
```

The following program shows the single inheritance using protected derivation

```
#include<conio.h>
#include<iostream.h>
class worker                                        //Base class declaration
{ protected:
int age; char name [20];
public:
void getworker( );
void showworker( );
};
void worker :: getworker( )
{
cout >> “your name please”;
cin >> name;
cout << “your age please”;
cin >> age;
}
void worker :: showworker( )
{
cout << “in my name is: “ << name << “in my age is “ <<age;
}
class manager:: protected worker                    // protected inheritance
{
int now;
public:
void getnow( );
void shownow( ) ;
};
```

```
void manager : : getnow( )
{
cout << “please enter the name In”;
cin >> name;
cout<< “please enter the age In”; //Directly inputting the data
cin >> age; members of base class
cout << “ please enter the no. of workers under you:”;
cin >> now;
}
void manager : : shownow( )
{
cout « "your name is : "«name«" and age is : "«age;
cout «“No. of workers under your are : "«now;

main ( )
{
clrscr ( ) ;
manager ml;
ml.getnow( ) ;
cout « "\n \n";
ml.shownow( );
}
```

Making a Private Member Inheritable

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

a. **Private:** when a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class and therefore the public members of the base class can be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derived class.

b. **Public:** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.

c. **Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these two classes.

The below mentioned table summarizes how the visibility of members undergo modifications when they are inherited

Base Class Visibility	Derived Class Visibility		
	Public	Private	Protected
Private	X	X	X
Public	Public	Private	Protected
Protected	Protected	Private	Protected

The private and protected members of a class can be accessed by:

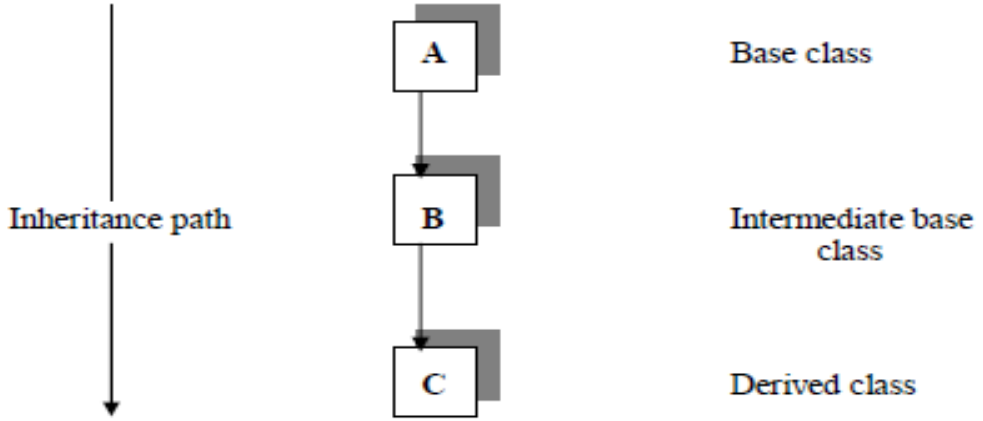
a. A function i.e. friend of a class.

b. A member function of a class that is the friend of the class.

c. A member function of a derived class.

Multilevel Inheritance

When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called 'MULTILEVEL INHERITENCE'. The class B is known as the 'INTERMEDIATE BASE CLASS' since it provides a link for the inheritance between A and C. The chain ABC is called 'ITNHERITENCE*PATH' for e.g.



The declaration for the same would be:

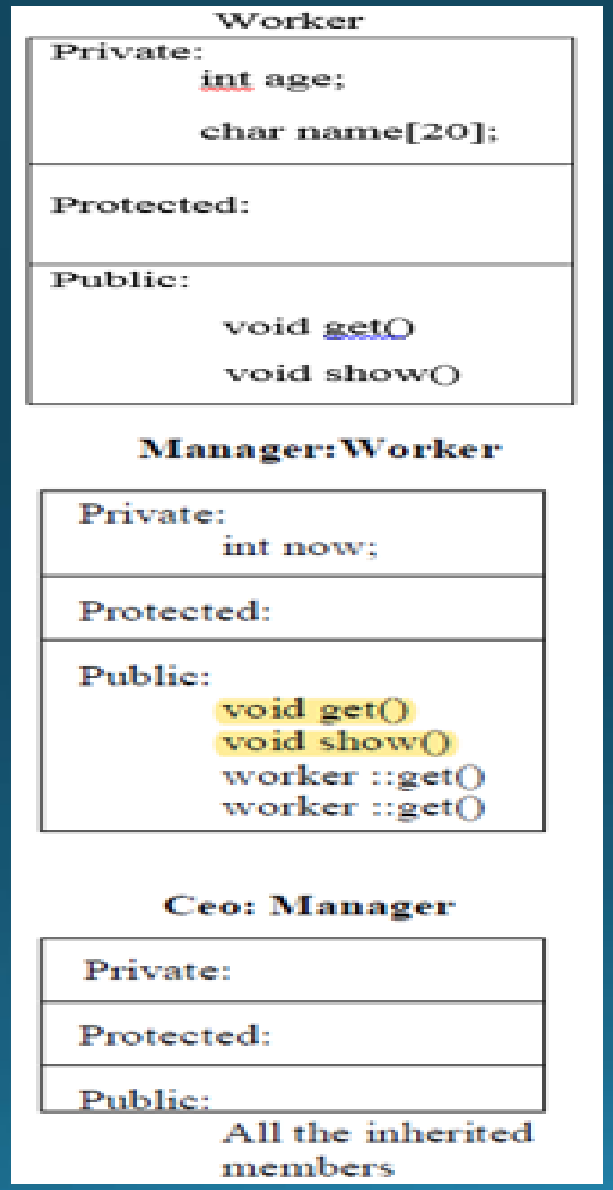
```
Class A
{
//body
}
Class B : public A
{
//body
}
Class C : public B
{
//body
}
```

This declaration will form the different levels of inheritance.
Following program exhibits the multilevel inheritance.

```
#include<iostream.h>
#include<conio.h>
class worker                                        // Base class declaration  - worker
{
int age;
char name [20] ;
public;
void get( ) ;
void show( ) ;
}
void worker: get ( )
{
cout << “your name please” ;
cin >> name;
cout << “your age please” ;
}
void worker : : show ( )
{
cout << “In my name is : “ <<name<< “ In my age is : “ <<age;
}
class manager : public worker                       //Intermediate base class derived - manager
{ //publicly from the base class
int now;
public:
void get ( ) ;
void show( ) ;
};
```

```
void manager :: get ( )
{
worker : :get () ; //calling get ( ) fn. of base class
cout << “no. of workers under you:”;
cin >> now;
}
void manager : : show ( )
{
worker : : show ( ) ; //calling show ( ) fn. of base class
cout << “In no. of workers under me are: “<< now;
}
class ceo: public manager                    //publicly inherited from the intermediate base class - ceo
{
int nom;
public:
void get ( ) ;
void show ( ) ;
};
void ceo : : get ( )
{
manager : : get ( ) ;
cout << “no. of managers under you are:”; cin >> nom;
}
 void manager : : show ( )
{
cout << “No. of managers under me are: “<<nom;
}
```

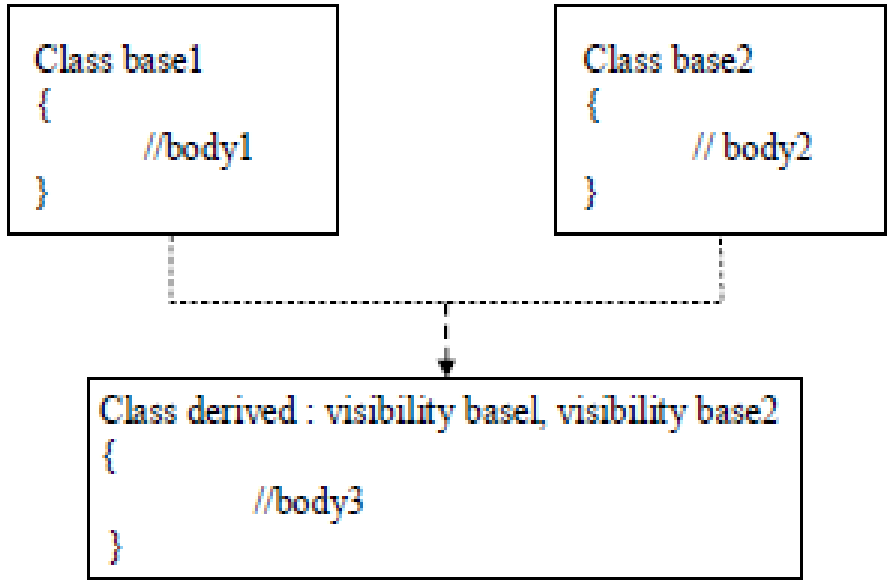
```
main ( )
{
clrscr ( ) ;
ceo cl ;
cl.get ( ) ; cout << "\n\n";
cl.show ( ) ;
}
```



Multiple Inheritances

A class can inherit the attributes of two or more classes. This mechanism is known as 'MULTIPLE INHERITENCE'. Multiple inheritance allows us to combine the features

of several existing classes as a starring point for defining new classes. It is like the child inheriting the physical feature of one parent and the intelligence of another. The syntax of the derived class is as follows:



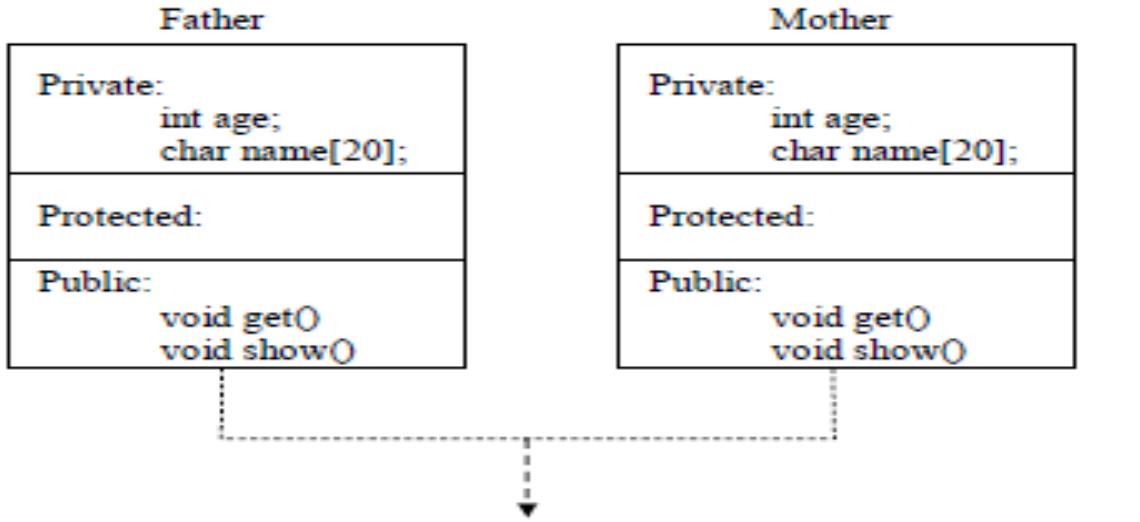
Where the visibility refers to the access specifiers i.e. public, private or protected. Following program shows the multiple inheritance.

```
#include<iostream.h>
#include<conio . h>
class father                                                    //Declaration of base classl
{
int age ;
char name [20] ;
public:
void get ( ) ;
void show ( ) ;
};
void father : : get ( )
{
cout << “your father name please”;
cin >> name;
cout << “Enter the age”;
cin >> age;
}
void father : : show ( )
{
cout<< “My father’s name is: ‘ <<name<< “My father’s age is:<<age;
}
```

```
class mother                                                    //Declaration of base class 2
{
char name [20] ;
int age ;
public:
void get ( )
{
cout << “mother’s name please” ;
cin >> name;
cout << “mother’s age please” ;
cin >> age;
}
void show ( )
{
cout << “My mother’s name is: “ <<name;
cout << “My mother’s age is: “ <<age;
}
class daughter : public father, public mother   //derived class inheriting publicly the features of both the base class
{
char name [20] ;
int std;
public:
void get ( ) ;
void show ( ) ;
};
```

```
void daughter :: get ( )
{
father :: get ( ) ;
mother :: get ( ) ;
cout << “child's name: “;
cin >> name;
cout << “child's standard”;
cin >> std;
}
void daughter :: show ( )
{
father :: show ( );
mother :: show ( ) ;
cout << “In child’s name is : “ <<name;
cout << “In child's standard: “ << std;
}
main ( )
{
clrscr ( ) ;
daughter d1;
d1.get ( ) ;
d1.show ( ) ;
}
```

Diagrammatic Representation of Multiple Inheritance is as follows:



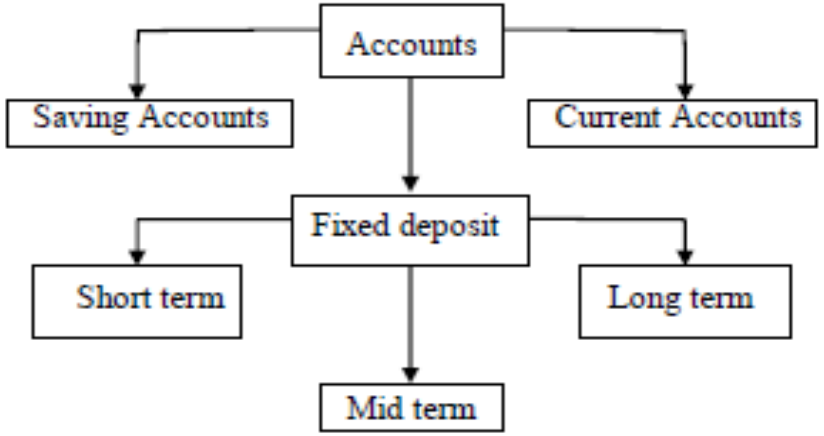
Class daughter: public Father, public Mother

Private: char name[20]; int age;
Protected:

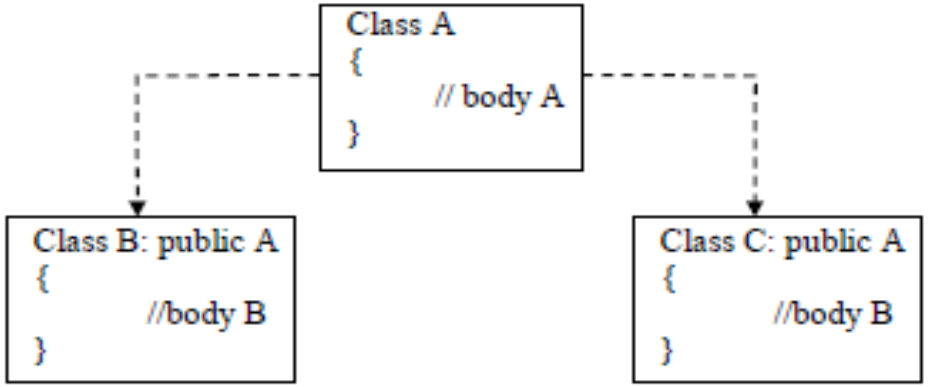
Public:
//self
void get(); void showQ;
//from Father
void get(); void show();
//from Mother
void get(); void show();

Hierarchical Inheritance

Another interesting application of inheritance is to use is as a support to a hierarchical design of a class program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level for e.g.



In general the syntax is given as



In C++, such problems can be easily converted into hierarchies. The base class will include all the features that are common to the subclasses. A sub-class can be constructed by inheriting the features of base class and so on.

```
// Program to show the hierarchical inheritance
 #include<iostream.h>
# include<conio. h>
class father                          //Base class declaration
{
int age;
char name [15];
public:
void get ( )
{
cout<< “father name please”; cin >> name;
cout<< “father’s age please”; cin >> age;
}
void show ( )
{
cout << “father’s name is ‘: “<<name;
cout << “father’s age is: “<< age;
}
};
class son : public father                  //derived class 1
{
char name [20] ;
int age ;
public;
void get ( ) ;
void show ( ) ;
} ;
```

```
void son : : get ( )
{
father :: get ( ) ;
cout << “your (son) name please” ; cin >>name;
cout << “your age please” ; cin>>age;
}
void son :: show ( )
{
father : : show ( ) ;
cout << “my name is : “ <<name;
cout << “my age is : “ <<age;
}
class daughter : public father                    //derived class 2.
{
char name [15] ;
int age;
public:
void get ( )
{
father : : get ( ) ;
cout << “your (daughter’s) name please”;  cin>>name;
cout << “your age please”; cin >>age;
}
```

```
void show ( )
{
father : : show ( ) ;
cout << “my name is: “ << name;
cout << “my age is: “ <<age;
}
};
main ( )
{
clrscr ( ) ;
son S1;
daughter  D1 ;
S1. get ( ) ;
D1. get ( ) ;
S1 .show( ) ;
D1. show ( ) ;
}
```

Hybrid Inheritance

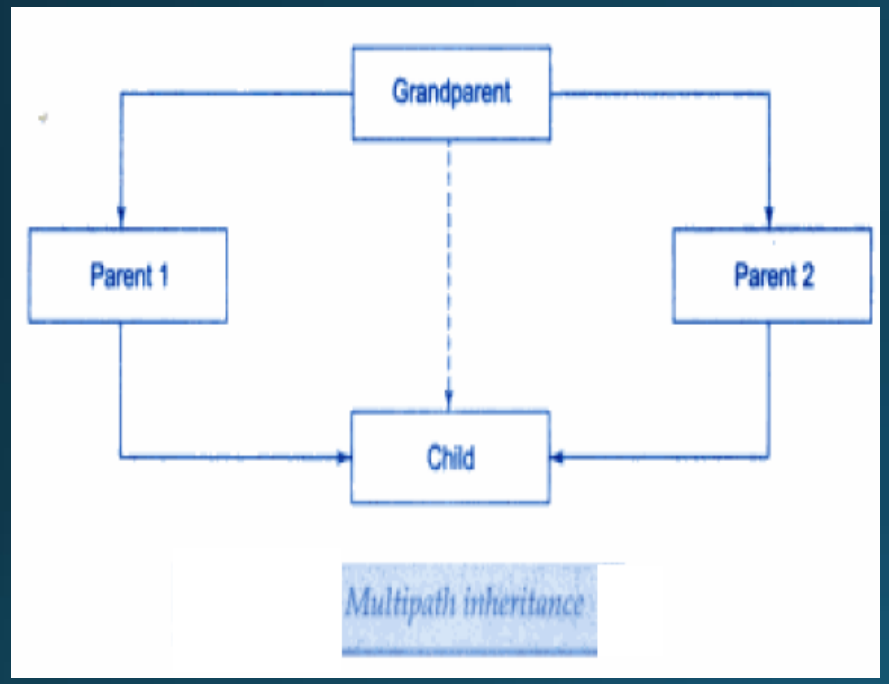
There could be situations where we need to apply two or more types of inheritance to design a program. Basically Hybrid Inheritance is the combination of one or more types of the inheritance. Here is one implementation of hybrid inheritance.

```
//Program to show the simple hybrid inheritance
#include<i sos t ream. h>
#include<conio . h>
class student                          //base class declaration
{
protected:
int r_no;
public:
void get _n (int a)
{
r_no =a;
}
void put_n (void)
{
cout << “Roll No. : “<< r_no;
cout << “\n”;
}
};
```

```
class test : public student              //Intermediate  base class
{
protected : int part1, part2;
public :
void get_m (int x, int y)
{  parti = x; part 2 = y;  }
void put_m (void) {
cout << "marks obtained: " << "In"
<< "Part 1 = " << part1 << "in"
<< "Part 2 = " << part2 << "In";
}
};
class sports                                          // base for result
{
protected : int score;
public:
void get_s (int s)
 { score = s }
void put_s (void)
{ cout << " sports wt. : " << score << "\n\n";  }
};
class result : public test, public sports          //Derived from test & sports
{
int total;
public:
void display (void);
};
```

```
void result : : display (void)
{
total = part1 + part2 + score;
put_n ( ) ;.
put_m ( );
put_S ( );
cout << “Total score: “ <<total<< “\n”
}
main ( )
{
clrscr ( ) ;
result S1;
S1.get_n (347) ;
S1.get_m (30, 35);
S1.get_s (7) ;
S1.dciplay ( ) ;
}
```

Virtual Base Classes



The duplication of the inherited members can be avoided by making common base class as the virtual base class: for e.g.

```
class g_parent
{
//Body
};
class parent1: virtual public g_parent
{
// Body
};
class parent2: public virtual g_parent
{
// Body
};
class child : public parent1, public parent2
{
// body
};
```

The 'child' has two direct base classes ‘parent1’ and ‘parent2’ which themselves has a common base class ‘grandparent’.

The child inherits the traits of ‘grandparent’ via two separate paths. It can also be inherit directly as shown by the broken line.

The grandparent is sometimes referred to as ‘INDIRECT BASE CLASS’.
Now, the inheritance by the child might cause some problems.

All the public and protected members of ‘grandparent’ are inherited into ‘child’ twice, first via ‘parent1’ and again via ‘parent2’.
So, there occurs a duplicacy which should be avoided.

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between virtual base class and derived class.

The keywords ‘virtual’ and ‘public’ can be used in either order.

```
//Program to show the virtual base class
#include<iostream.h>
#include<conio . h>
class student // Base class declaration
{ protected: int r_no; public:
void get_n (int a)
{ r_no = a; }
void put_n (void)
{ cout << “Roll No. “ << r_no<< “ln”;}
};
class test : virtual public student // Virtually declared common { //base
class 1
protected:
int part1;
int part2;
public:
void get_m (int x, int y)
{ part1= x; part2=y;}
void putm (void)
{
cout << “marks obtained: “ << “\n”;
cout << “part1 = “ << part1 << “\n”;
cout << “part2 = “<< part2 << “\n”;
}
};
```

```
class sports : public virtual student // virtually declared common { //base class 2
protected:
int score;
public:
void get_s (int a) {
score = a ;
}
void put_s (void)
{ cout << “sports wt.: “ <<score<< “\n”;}
};
class result: public test, public sports //derived class
{
private : int total ;
public:
void show (void) ;
};
void result : : show (void)
{ total = part1 + part2 + score ;
put_n ( );
put_m ( );
put_s ( ) ; cout << “\n total score= “ <<total<< “\n” ;
}
```

```
main ( )
{
clrscr ( ) ;
result S1 ;
S1.get_n (345)
S1.get_m (30, 35) ;
S1.get-S (7) ;
S1. show ( ) ;
}
```

```
//Program to show hybrid inheritance using virtual base classes
#include<iostream.h>
#include<conio.h>
Class A
{
protected:
int x; public:
void get (int) ;
void show (void) ;
};
void A : : get (int a)
{
x = a ;
}
void A : : show (void)
{
cout << X ;
}
```

```
Class A1 : Virtual Public A
{
protected:
int y ;
public:
void get (int) ;
void show (void);
};
void A1 :: get (int a)
{ y = a;}
void A1 :: show (void)
{
cout <<y ;
{
class A2 : Virtual public A
{
protected:
int z ;
public:
void get (int a)
{ z =a;}
void show (void)
{ cout << z;}
};
```

```
class A12 : public A1, public A2
{
int r, t ;
public:
void get (int a)
{ r = a;}
void show (void)
{ t = x + y + z + r ;
cout << “result =” << t ;
}
};
main ( )
{
clrscr ( ) ;
A12 r ;
r.A : : get (3) ;
r.A1 : : get (4) ;
r.A2 : : get (5) ;
r.get (6) ;
r . show ( ) ;
}
```

Abstract Classes

An *abstract class* is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes). It is a design concept in program development and provides a base upon which other classes may be built.

Member Classes: Nesting of Classes

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below:

```
class alpha {....};
class beta {....};
class gamma
{
        alpha a;            // a is an object of alpha class
        beta b;             // b is an object of beta class
        .....
};
```

All objects of **gamma** class will contain the objects **a** and **b**. This kind of relationship is called *containership* or *nesting*. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other 'ordinary' members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```
class gamma
{
        .....
        alpha a;          // a is object of alpha
        beta b;           // b is object of beta
  public:
        gamma(arglist): a(arglist1), b(arglist2)
        {
                  // constructor body
        }
};
```

arglist is the list of arguments that is to be supplied when a **gamma** object is defined. These parameters are used for initializing the members of **gamma**. *arglist1* is the argument list

for the constructor of **a** and *arglist2* is the argument list for the constructor of **b**. *arglist1* and *arglist2* may or may not use the arguments from *arglist*. Remember, **a**(*arglist1*) and **b**(*arglist2*) are function calls and therefore the arguments do not contain the data types. They are simply variables or constants.

Example:

```
gamma(int x, int y, float z) : a(x), b(x,z)
{
        Assignment section(for ordinary other members)
}
```

We can use as many member objects as are required in a class. For each member object we add a constructor call in the initializer list. The constructors of the member objects are called in the order in which they are declared in the nested class.

REFERENCES:

1.E. Balagurusamy, “Object Oriented Programming with C++”, Fourth edition, TMH, 2008.

2. LECTURE NOTES ON Object Oriented Programming Using C++ by Dr. Subasish Mohapatra, Department of Computer Science and Application College of Engineering and Technology, Bhubaneswar Biju Patnaik University of Technology, Odisha

3. K.R. Venugopal, Rajkumar, T. Ravishankar, “Mastering C++”, Tata McGraw-Hill Publishing Company Limited

4. Object Oriented Programming With C++ - PowerPoint Presentation by Alok Kumar

5. OOPs Programming Paradigm – PowerPoint Presentation by an Anonymous Author