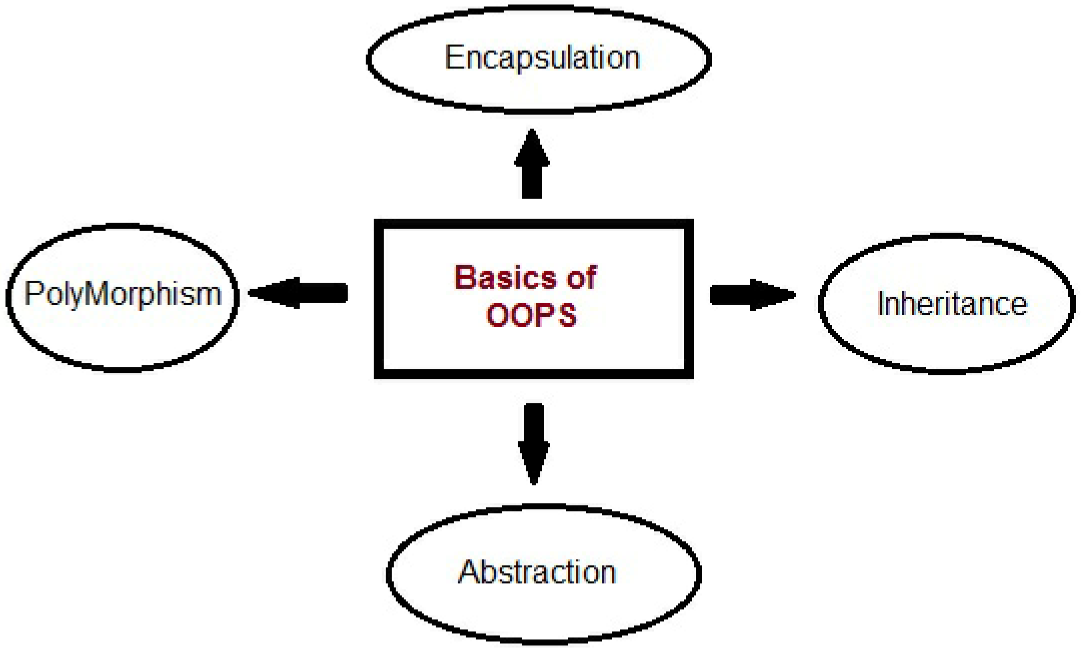
C++ and Object Oriented Programming

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance

Objects are instances of classes and are used to interact amongst each other to create applications. Instance means, the object of class on which we are currently working.



For Example : We consider human body as a class, we do have multiple objects of this class, with variable as color, hair etc. and methods as walking, speaking etc.

Some of the main features of object oriented programming which you will be using in C++.

1. Objects
2. Classes
3. Abstraction
4. Encapsulation
5. Inheritance
6. Overloading
7. Exception Handling

Objects

Objects are the basic unit of OOP. They are instances of class, which have data members and uses various member functions to perform tasks.

Class

It is similar to structures in C language. Class can also be defined as user defined data type but it also contains functions in it. So, class is basically a blueprint for object. It declare & defines what data variables the object will have and what operations can be performed on the class's object.

Abstraction

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes provide methods to the outside world to access & use the data variables, but the variables are hidden from direct access.

Encapsulation

It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class.

Inheritance

Inheritance is a way to reuse once written code again and again. The class which is inherited is called base calls & the class which inherits is called derived class. So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

Polymorphism

Polymorphion makes the code more readable. It is a features, which lets is create functions with same name but different arguments, which will perform differently. That is function with same name, functioning in different

Overloading

Overloading is a part of polymorphion. Where a function or operator is made & defined many times, to perform different functions they are said to be overloaded.

Exception Handling

Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime.

Object Oriented Programming has great advantages over other programming styles:

1. Code Reuse and Recycling: Objects created can easily be reused in other programs.
2. Encapsulation (part 1): Once an Object is created, knowledge of its implementation is not necessary for its use. In older programs, coders needed understand the details of a piece of code before using it .
3. Encapsulation (part 2): Objects have the ability to hide certain parts of themselves from programmers. This prevents programmers from tampering with values they shouldn't.
4. Design Benefits: Large programs are very difficult to write. Object Oriented Programs force designers to go through an extensive planning phase, which makes for better designs with less flaws.
5. Software Maintenance: An Object Oriented Program is much easier to modify and maintain. So although some additional work is spent before the program is written, less work is needed to maintain it over time.

Defining Class and Declaring Objects

When we define any class, we do not defining any data, but we just define a structure, as to what the object of that class type will contain and what operations can be performed on that object.

Below is the syntax of class definition,

```
class ClassName
{
 Access specifier:
                Data members;
                MemberFunctions()
                       {
                       }
};
```

Here is an example, we have made a simple class named Student with appropriate members,

```
class Student
{
 public:
        int rollno;  string name;
};

int main()

{

 Student A;

 Student B;

}
```

Here A and B are the objects of class Student, declared with the class definition.

Access Control in Classes

Access specifiers in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

public   private   protected

These access specifiers are used to set boundaries for availability of members of class be it data members or member functions

Public

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public.

```
class A

{

 public:   // public access specifier

        int x;          // Data Member Declaration
        void display();   // Member Function decaration
}
```

Private

Private keyword, means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.

```
class B
{
 private:   // private access specifier
        int x;          // Data Member Declaration
        void display();   // Member Function decaration
}
```

Protected

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class.

```
class C
{
 protected:   // protected access specifier
          int x;          // Data Member Declaration
        void display();   // Member Function decaration
}
```

Constructors

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after memory space is allocated to the object.

```
class Test
{
 int x;
 public:
       Test();  //Constructor
};
Test::Test()   // Constructor expansion
{
  x=1;
}
```

While defining a contructor we must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.

Types of Constructors

Constructors are of three types :

1. Default Constructor   2. Parametrized Constructor   3. Copy COnstructor

Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax :

```
class_name ()
   {
    Constructor Definition
  }
```

Example :

```
class Cube
{
       int side;
public:
   Cube()
     {
        side=10;
```

```
 }
};


Void  main()
{
Cube c;
cout << c.side;
}
```

Output : 10

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
 int side;
};

void main()
{
 Cube c;
 cout << c.side;
}
```

Output : 0

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 in this case.

Parameterized Constructor

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

Example :

```
class Cube
{
 int side;
 public:
  Cube(int x)
  {
   side=x;
  }
};

void main()
```

```
{
 Cube c1(10);
 Cube c2(20);
 Cube c3(30);
 cout << c1.side;
 cout << c2.side;
 cout << c3.side;
}
```

OUTPUT : 10 20 30

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

Copy Constructor

These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object.

Constructor Overloading

Just like other member functions, constructors can also be overloaded. In fact when we have both default and parameterized constructors defined in a classwe are having Overloaded Constructors, one with no parameter and other with parameter.

we can have any number of Constructors in a class that differ in parameter list.

```
class Student
{
 int rollno;
 string name;

 public:

 Student(int x)
 {
  rollno=x;
  name="None";
 }

 Student(int x, string str)
 {
  rollno=x ;
  name=str ;
 }
};

void main()
{
 Student A(10);
 Student B(1001,"Pushkal");
}
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

Destructors

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** `~` sign as prefix to it.

```
class Test

{

 public:

        ~Test();

};
```

Destructors will never have any arguments.

Example to show how Constructor and Destructor is called

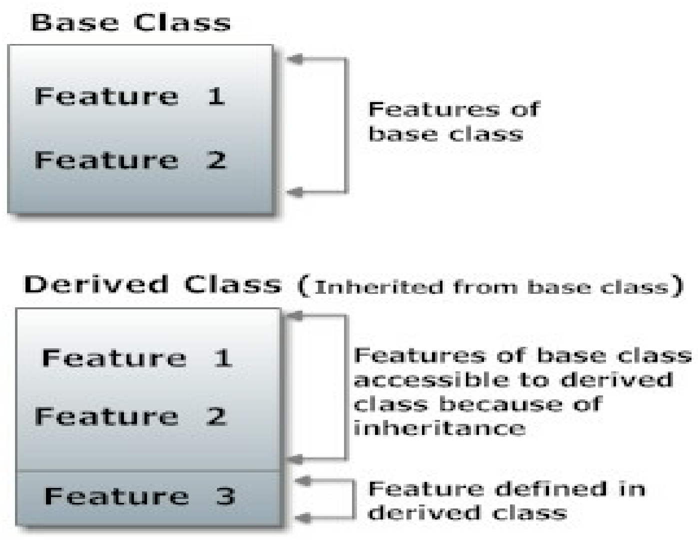
```
class Test
{
Test()
 {
  cout << "Constructor called";
 }

~Test()
 {
  cout << "Destructor called";
 }
};

void  main()
{
 Test   z1;   // Constructor Called for object z1
 int x=1;

 if(x)
  {
```

```
  Test z2;  // Constructor Called for object z2
 }   // Destructor Called for obj  z2
} //  Destructor called for obj z1
```

Inheritance

Inheritance allows a programmer to define a new class in terms of an existing class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

The derived class inherits all feature from a base class and it can have additional features of its own.


Base Class
Feature 1
Feature 2
Features of base class
Derived Class (Inherited from base class)
Feature 1
Feature 2
Feature 3
Features of base class accessible to derived class because of inheritance
Feature defined in derived class

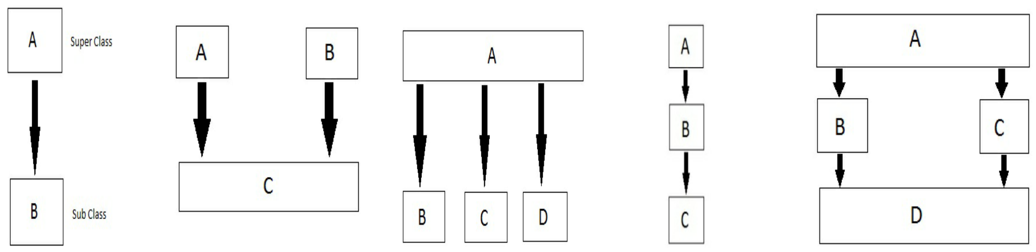
This existing class is called the base class, and the new class is referred to as the derived class.

The idea of inheritance implements the **is a** relationship. In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.

Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.

Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.

Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.

Syntax of Single inheritance

```
class    derived-class :    access-specifier    base-class
```

Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.
Example

```
class Rectangle
{
  ... .. ...
};
class Area : public Rectangle
{
  ... .. ...
};

class Perimeter : public Rectangle
{
  .... .. ...
};
```

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class

members that should not be accessible to the member functions of derived classes should be declared private in the base class.
We can summarize the different access types in the following way:

Access	**Public**	**protected**	**private**
Same class	Yes	yes	yes
Derived classes	Yes	yes	no
Outside classes	Yes	no	no

A derived class inherits all base class methods with the following **exceptions**:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

- **Public Inheritance:** When deriving a class from a **public** base class,**public** members of the base class become **public** members of the derived class and **protected** members of the base class become**protected** members of the derived class. A base class's **private**members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

- **Protected Inheritance:** When deriving from a **protected** base class,**public** and **protected** members of the base class become **protected**members of the derived class.

- **Private Inheritance:** When deriving from a **private** base class,**public** and **protected** members of the base class become **private**members of the derived class.

Example :

```
class A
{
public:
        int x;
protected:
        int y;
private:
        int z;
};

class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};

class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

IMPORTANT NOTE: Classes B, C and D all contain the variables x, y and z. It is just question of access.

Syntax of Multiple Inheritance:
A C++ class can inherit members from more than one class and here is the extended syntax:

```
class   derived-class  : access    baseA,    access    baseB....
```

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above.

Constructor/destructor under inheritance

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

1. Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.
2. To call base class's parameterised constructor inside derived class's parameterised constructor, we must mention it explicitly while declaring derived class's parameterized constructor.

Whenever we kill the derived class object, first the derived class destructor is executed and then the parent class's destructor finishes execution.

Friend Functions in C++:-

It is a type of a function which has the permission to access the private and protected members of a class. Normally private and protected members of a class could not be accessed from outside the class but in some cases the program may have to access the private or protected variables to perform the functionality. In that case we declare a function in a class with a friend keyword whose private members are required to access by the program. Therefore, this function will be able to access s such members .

Two classes having a common friend function

class B; // prior delcaration

```
class  A
{
private:
        int a;
public:
A()
{
a=25;
}
friend void show(A,B);
};


class   B
{
private:
         int b;
public:
B()
{
```

```
b=35;
}
friend void show(A,B);
};

void show(A  x,  B  y)
{
int r;
r= x.a + y.b;
cout<<”The value of class A object =”<<x.a<<endl;
cout<<”The value of class B object =”<<y.b<<endl;
cout<<”The sum of both values =”<<r<<endl;
}

void main()
{
A    obj1;
B    obj2;
show(obj1, obj2);
getch();
}
```

In the above program, first of all we have declared a class B, we are just declaring it, so when we write a signatures of a friend function, we will use both classes A and B so that the compiler do not object.

In a class we will declare one integer type private data member and initialize it to a value 25 in a constructor of A. Then we will define a function with the friend keyword which contains parameters as both classes. Then we will write the definition of class B. This class will also declares a one private data member and initialize it to a value 35 in a constructor of class B.

Then it will declare a function with a friend keyword whose signatures are both classes. Then we will write the definition of a friend function show, in which we will show the value of the private data members of both classes. Then in main function we will simply create the objects of these classes and call the friend function Show to display their respective fields.

Friend classes in C++:-

These are the special type of classes whose all of the member functions are allowed to access the private and protected data members of a particular class is known as friend class. Normally private and protected members could not be accessed from outside the class but in some situations the program has to access them in order to produce the desired results. In such situations the friend classes are used, the use of friend classes allows a class to access these members of another class.

The syntax of declaring a friend class is not very tough, simply if a class is declared in another class with a friend keyword it will become the friend of this class. The below example will completely illustrate the use of a friend classes.

```
class A
{
friend class B;
private:
        int a;
public:
A()
{
a=10;
}
};


class B
{
public:

void show(A  obj)
{
cout<<"The value of a: "<<obj.a<<endl;
}
};

void main()
{
A  x;
B  y;
y.show(x);
getch();
}
```

In the above example we have declared two classes A and B. In A we simply declared an integer type private data member and initialized them in a constructor. In this class we have also declared a class B with a friend keyword which makes B the friend of class A. Therefore all the member functions of B will be able to access the private data members of A. Then in class B we simply created a function show() which have a permission to access the private members of class A. Then we will display the values of the attributes of the object of class A.

Function Overriding

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**

Requirements for Overriding

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Example of Function Overriding

```
class Base
{
 public:
 void show()
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}
```

In this example, function **show()** is overridden in the derived class

Pointers to class members

Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.

Defining a pointer of class type

We can define pointer of class type, which can be used to point to class objects.

```
class Simple
{
 public:
 int a;
};

int main()
{
 Simple   z;
 Simple  *p;   // Pointer of class type
 p = &z;

 cout << z.a;
 cout << p->a;  // Accessing member with pointer
}
```

In the above example we can see the declaration of a pointer of class type which points to class's object.

We can access data members and member functions using pointer name with arrow -> symbol.

What is this pointer?

Every object has a special pointer called "**this**" which points to the very object itself.

This pointer is accessible to all members of the class but not to any static members of the class.

this pointer can be used to find the address of the object in which the function is a member.

```
Class  Test {
        public:
                  int  d;
        Test()
             {    d=100;   };
        void   Print1();
        void   Print2();
};

// with out  using this pointer
void  Test  :: Print1() {
    cout << d << endl;
}

// By Using this pointer
void  Test::Print2() {
    cout << "My address = " << this << endl;
    cout << this->d << endl;
}


void main()
{
    Test    a;
    a.Print1();
    a.Print2();

}
```

Static binding (compile time binding)

If a base class and derived class has same function and if you write code to access that function using pointer of base class then, the function in the base class is executed even if, the object of derived class is referenced with that pointer variable.

This can be demonstrated by an example.

```
#include <iostream>

class B
{
    public:
       void display()

         { cout<<"Content of base class.\n"; }
};

class D : public B
{
    public:
       void display()

         { cout<<"Content of derived class.\n"; }
};

void main()
{
    B   *b;
    D   d;

    b = &d;    /* Address of object d  given to  parent class pointer variable b */
    b->display();

}
```

Output

```
Content of base class.
```

In above program, even if the object of derived class d is put in pointer to base class, display( ) of the base class is executed. The reason is static binding or early binding of the methods to the name of the parent pointer at compile time itself.

Dynamic binding or late binding (run time polymorphism)

If you want to execute the member function of derived class then, you can declare display( ) in the base class as virtual which makes that functions binding to be postponed till runtime.

```
/* Example to demonstrate the working of virtual function in C++ programming. */

#include <iostream>

class B
{
   public:
    virtual void display()      /* Virtual function */

       { cout<<"Content of base class.\n"; }
};

class D1 : public B
{
   public:
     void display()
       { cout<<"Content of first derived class.\n"; }
};


class D2 : public B
{
   public:
     void display()
       { cout<<"Content of second derived class.\n"; }
};

void main()
{
   B   *b;
   D1   d1;
   D2   d2;
   b = &d1;
   b->display();   /* calls display() of class derived D1 */


   b = &d2;
   b->display();   /* calls display() of class derived D2 */
   return 0;
}
```

Output

```
Content of first derived class.
Content of second derived class.
```

After the function of base class is made virtual, the code b->display( ) will call the display( ) of the appropriate derived class depending upon the content of pointer.

Pure virtual function, Abstract class

If expression =0 is added to a virtual function then, that function is becomes pure virtual function. Note that, adding =0 to virtual function does not assign value, it simply indicates the virtual function is a pure function.

If a base class contains at least one virtual function then, that class is known as **Abstract class**.

```
#include <iostream>

class Shape                 /* Abstract class */
{
   protected:
```

```
                float   l;
   public:
      void get_data()         /* Note: this function is not virtual. */
      {
         cin>>l;
      }

      virtual float area() = 0; /* Pure virtual function */
};

class Square : public Shape
{
   public:
         float area()
                   {   return    l*l;    }
};

class Circle : public Shape
{
   public:
             float area()
                        { return   3.14*l*l;   }
};

void   main()
{
   Square   s;
   Circle   c;
   cout<<"Enter length to calculate area of a square: ";
   s.get_data();
   cout<<"Area of square: "<<s.area();
   cout<<"\nEnter radius to calcuate area of a circle:";
   c.get_data();
   cout<<"Area of circle: "<<c.area();

}
```

In this program, pure virtual function virtual float area() = 0; is defined inside class Shape, so this class is an abstract class and one cannot create object of class Shape.

Virtual Destructors

Destructors in the Base class can be Virtual. Whenever Upcasting is done, Destructors of the Base class must be made virtual for proper destrucstion of the object when the program exits.

NOTE :   Constructors are never Virtual, only Destructors can be Virtual.

Lets first see what happens when we do not have a virtual Base class destructor.

```
class Base
{
 public:
 ~Base()
        {         cout << "Base Destructor\t";   }
};

class Der  :  public Base
{
 public:
                ~Der()
                         { cout<< "Derived Destructor"; }
};

int main()
{
 Base* b = new  Der;     //Upcasting
 delete b;
}
```

Output :   Base Destructor

In the above example, **delete b** will only call the Base class destructor, which is undesirable because, then the object of Derived class remains undestructed, because its destructor is never called. Which results in memory leak. We must take care to use virtual destructor concept during **Upcasting .**

Now lets see. what happens when we have Virtual destructor in the base class.

```
class Base
{
 public:
                 virtual   ~Base()
                            {cout << "Base Destructor\t"; }
};

class Der  :  public Base
{
 public:
        ~Der()
                        { cout<< "Derived Destructor"; }
};

int main()
{
 Base* b = new Der;     //Upcasting
 delete b;
}
```

Output :

```
Derived Destructor
Base Destructor
```

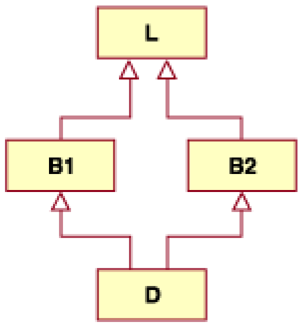
When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behaviour.

Virtual Base Classes

Suppose there are two derived classes B and C that have a common base class A, and also there is another class D that inherits from B and C. We can then declare the base class A as *virtual* to ensure that B and C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects of class L, one through class B1 and another through class B2. We can use the keyword **virtual** in front of the base class specifiers in the *base lists* of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

For example:



```
class L
   {
     /* ... */
   }; // indirect base class

class B1 : virtual public L
    {
     /* ... */
    };

class B2 : virtual public L
    {
    /* ... */
    };

class D : public B1, public B2
   {
     /* ... */
    }; // valid
```

Using the keyword **virtual** in this example ensures that an object of class D inherits only one subobject of class L

function templates

Template is a feature which provides support for generic programming. Template allows creation of generic type elements. For e.g, if we define a function which takes int type parameters, the function can operate on integer elements only but using templates, we can define a generic function which can operate on any data type.

Templates can be applied to functions and classes.

Function Templates

In the following program, we implement a swap( ) function which can perform swapping of two elements of any data type

```
template <class T>
void    swapValues(T   &a, T   &b)
 {
   T    temp;
   temp = a;
   a = b;
   b = temp;
}
```

```
void main() {
  int a = 5, b = 7;
  float x = 65.39, y = 27.89;
  char p = 'A', q = 'B';

  cout << "Before Swapping :- " << endl;
  cout << "a = " << a << "  b = " << b << endl;
  cout << "x = " << x << "  y = " << y << endl;
  cout << "p = " << p << "  q = " << q << endl;

  /* swap the values stored in variables */
  swapValues(a, b); // swap two integer values
  swapValues(x, y); // swap two float values
  swapValues(p, q); // swap two character values

  cout << "\nAfter Swapping :- " << endl;
  cout << "a = " << a << "  b = " << b << endl;
  cout << "x = " << x << "  y = " << y << endl;
  cout << "p = " << p << "  q = " << q << endl;

  }
```

Class Templates

Consider a class which supports three relational operations (getMax(), getMin() and isEqual()). Following program makes the class generic using class template to perform the operations on different data types :

```
/* Define a generic class to perform relational operation */
template <class T>
class relational
 {
  T   x, y;
public :

  relational(T    var1, T    var2)
  {
     x = var1;
     y = var2;
  }
  T   getMin() {
     return ((x < y) ? x : y);
  }
  T   getMax() {
     return ((x > y) ? x : y);
  }
};
```

```
void main()
 {
   relational<int>    r1(11, 17); // object r1 with integer data members
   relational<double>     r2(7.1, 6.9); // 'T' is replaced with 'double' type

   int    min = r1.getMin(); // get min value of object r1
   double    max = r2.getMax(); // get max value of object r2

   cout << "Minimum Element of object r1 : " << min << endl;
   cout << "Maximum Element of object r2 : " << max << endl;
 }
```

Templates with Multiple Parameters

We can also define a template with multiple parameters. See the program below

```
// define a template class with two parameters ( T1 and T2 )

template <class T1, class T2>

class    X
 {
   T1    a;
   T2    b;
public :

    X(T1    var1, T2    var2) {
      a = var1;
      b = var2;
   }
   void display() {
      cout << "a : " << a << "  b : " << b << endl;
   }
};

void   main()
 {
   X<int, char>      obj1(7, 'M'); // T1 is integer and T2 is character
   X<float, int>    obj2(65.71, 8); // T1 is float and T2 is integer
```

```
  obj1.display();
  obj2.display();

}
```

Exception handling in C++

Exceptions are unusual conditions or problems that arises during the execution of a program like divide by zero, running out of memory or accessing an array out of its bounds. Exception Handling is a mechanism to handle such conditions at run time and prevent abnormal termination or crashing of the program.

Throwing and Catching Exceptions
We can throw an exception from any part of the program using throw keyword whenever an anomalous situation occurs at runtime and then appropriate action is taken after catching the exception.

The part of the code which is protected by exception handling is enclosed within try ... catch block. Following program illustrates the use of throw , try and catch keywords :

To catch exceptions we must place a portion of code under exception inspection. We can do this by putting that portion of code in a try block. So, when an exceptional circumstance arises an exception is thrown. This in turn transfers the control to the exception handler. If there are no exceptions thrown, the code will continue normally and all handlers are ignored.

```
float divide(float   num, float   denom) {
  if ( denom == 0 ) {
    throw "Divide by Zero Error";
  }
  else {
    return (num / denom);
  }
}

void main() {
  float x = 5, y = 0;
  /* protect the piece of code using try ... catch block
     and prevent crashing of the program */

try
 {
     float result = divide(x, y);
     cout << "Result : " << result << endl;
  }
 catch (const char *err_msg)
  {
     cout << "Exception Caught : " << err_msg << endl;
  }
}
```

In the above program, we have thrown an exception of type const char * and then caught it. We can have multiple catch blocks to catch different types of exception.

Operator Overloading

Operator Overloading is a feature of C++ through which one can assign special meanings to the operators. We can create new definition for the operators. For example, the addition operator (**+**) can be overloaded to work among two objects. All operators can be overloaded except some operators like scope resolution operator **(::)** , sizeof operator, conditional operator (**? :**) and class member access operator (.) , (**.***).

Example to Overload **unary –(minus**) operator

```
class   Test
 {
  int a, b;

public :
  Test()
        { }
  Test( int   var1,   int  var2)
  {
     a = var1;
     b = var2;
  }

 void   displayValues() {
     cout << "a : " << a << "  b : " << b << endl;
  }
  void   operator-()
       { // overload unary (-) operator
              a = -a;
              b = -b;
     }
 };
```

```
void    main()

 {

 Test    z(7, 2);
   z.displayValues();

  -z; // calling overloaded operator
   z.displayValues();

}
```

Output :
```
      a:    7       b:    2
       a:   -7       b:   -2
```

overloading binary + operator

The following example demonstrates how to overload + operator.

```
class   Test
 {
   int a, b;

public :
   Test()
          { }
   Test( int   var1,    int  var2)
   {
      a = var1;
      b = var2;
   }

 void   displayValues() {
      cout << "a : " << a << "  b : " << b << endl;
   }

Test   operator+(  Test   &x)
    {
      Test     temp_obj;
      temp_obj.a = this->a + x.a;
      temp_obj.b = this->b + x.b;
      return   temp_obj;
   }
  };

void  main()

{
   Test    z1(7, 2),    z2(9, 3);
   cout << "Displaying object1 values : ";
   z1.displayValues();
   cout << "Displaying object2 values : ";
   z2.displayValues();
   Test    sum_obj = z1 + z2; // call overloaded (+) operator
   cout << "After adding 2 objects : ";
   sum_obj.displayValues();

}
```

What is a copy constructor?

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
            ClassName (const ClassName     &old_obj);
```

Following is a simple example of copy constructor.

```
class Point
{
private:
   int x, y;
public:
   Point(int x1, int y1) { x = x1; y = y1; }

   // Copy constructor
   Point(const Point &p2) {x = p2.x; y = p2.y; }

   int getX()
         {  return x; }

   int getY()
         {  return y; }
};

void main()
{
   Point p1(10, 15); // Normal constructor is called here
   Point p2 = p1; // Copy constructor is called here

   // Let us access values assigned by constructors
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

   cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
}
```
Output:

```
p1.x = 10, p1.y = 15

p2.x = 10, p2.y = 15
```

In C++, a Copy Constructor may be called in following cases:
1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When compiler generates a temporary object.

File Handling

Files are a means to store data in a storage device. C++ file handling provides a mechanism to store output of a program in a file and read from a file on the disk. we know by using **<iostream>** header file it provide functions **cin** and **cout** to take input from console and write output to a console respectively. Now, we look at another header file **<fstream>** which provides data types or classes (**ifstream** , **ofstream** , **fstream**) to read from a file and write to a file.

File Opening Modes

A file can be opened in different modes to perform read and write operations. Function to open a file i.e **open( )**takes two arguments : **char *filename** and **ios :: mode**. C++ supports the following file open modes :

Mode	**Explanation**
ios :: in	Open a file for reading
ios :: out	Open a file for writing
ios :: app	Appends data to the end of the file
ios :: ate	File pointer moves to the end of the file but allows to writes data in any location in the file
ios :: binary	Binary File
ios :: trunc	Deletes the contents of the file before opening

If a file is opened in **ios :: out** mode, then by default it is opened in **ios :: trunc** mode also i.e the contents of the opened file is overwritten. If we open a file using **ifstream** class, then by default it is opened in **ios :: in** mode and if we open a file using **ofstream** class, then by default it is opened in **ios :: out** mode.
The **fstream** class doesn't provide any default mode.

File Operations using ifstream and ofstream

Open a file for writing

```
#include<iostream>
#include<fstream>

void main()
 {
   ofstream   ofile; // declaring an object of class ofstream
   ofile.open("file.txt"); // open "file.txt" for writing data

/* write to a file */
   ofile << "This is a line in a file" << endl;
   ofile << "This is another line" << endl;

   /* write to a console */
   cout << "Data written to file" << endl;
   ofile.close(); // close the file

}
```

Open a file for reading

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

void main() {
   char data[100]; // buffer to store a line read from file
   ifstream   ifile; // declaring an object of class ifstream

  ifile.open("file.txt"); // open "file.txt" for reading
   cout << "Reading data from a file :-" << endl << endl;

   while (!ifile.eof()) { // while the end of file [ eof() ] is not reached
      ifile.getline(data, 100); // read a line from file
      cout << data << endl; // print the file to console
   }

   ifile.close(); // close the file

}
```

In the first program, a file " file.txt " is created and some data is written into it. The file is created in the same directory in which the program file is saved.

In the second program, we read the file line by line using and then print each line on the console. The while loop continues till the end of file is reached. We ensure that using the condition while(! ifile.eof( ))