Inheritance

Inheritance allows us to define a class in terms of another 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.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

class derived-class: access-specifier base-class

Access specifiers:

***public** - members are accessible from outside the class. **private** - members cannot be accessed (or viewed) from outside the class. **protected** - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.*

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.

```
#include <iostream>


using namespace std;
```

```
// Base class

class Shape {

  public:

    void setWidth(int w) {

      width = w;

    }

    void setHeight(int h) {

      height = h;

    }


  protected:

    int width;

    int height;

};



// Derived class

class Rectangle: public Shape {

  public:
      int getArea() {

         return (width * height);

      }

};




int main(void) {

   Rectangle Rect;



   Rect.setWidth(5);

   Rect.setHeight(7);




   // Print the area of the object.

   cout << "Total area: " << Rect.getArea() << endl;




   return 0;

}
```

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.

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.

Multiple Inheritance

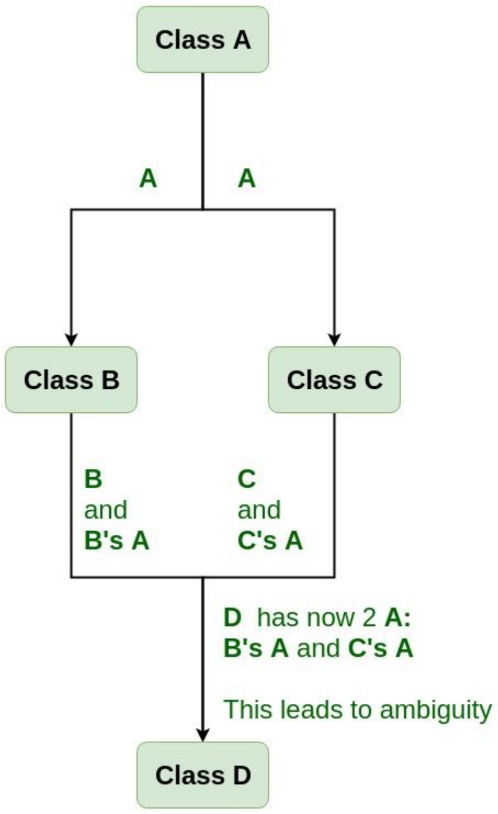
class derived-class: access baseA, access baseB....

```
#include <iostream>

using namespace std;

// Base class Shape
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

// Base class PaintCost
class PaintCost {
  public:
    int getCost(int area) {
      return area * 70;
    }
};

// Derived class
class Rectangle: public Shape, public PaintCost {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);
```

```
   area = Rect.getArea();

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   // Print the total cost of painting
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;

   return 0

}
```

Virtual base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.


Class A
A
A
Class B
Class C
B and B's A
C and C's A
D has now 2 A: B's A and C's A
This leads to ambiguity
Class D

As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

To show the need of Virtual Base Class in C++

```
#include <iostream>

using namespace std;


class A {
```

```
public:
    void show()
    {
        cout << "Hello form A \n";
    }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

int main()
{
    D object;
    object.show();
}
```

o/p:Compiler Error

After adding the virtual while inheriting A in B and C

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

class A {
public:
    void show()
    {
        cout << "Hello from A \n";
    }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
    D object;
    object.show();
}
```

Pure Virtual Functions and Abstract Classes in C++

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration

```
// An abstract class

class Test

{

   // Data members of class

public:

   // Pure Virtual Function

   virtual void show() = 0;


  /* Other members */

};
```

A pure virtual function is implemented by classes which are derived from a Abstract class.

```
#include<iostream>

using namespace std;



class Base

{

   int x;

public:

    virtual void fun() = 0;

    int getX() { return x; }

};



// This class inherits from Base and implements fun()

class Derived: public Base

{

    int y;

public:

    void fun() { cout << "fun() called"; }

};



int main(void)

{

    Derived d;

    d.fun();
    return 0;
}
```

A class is abstract if it has at least one pure virtual function.

We can have pointers and references of abstract class type.

Example:

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

class Base
{
public:
    virtual void show() = 0;
};

class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived();
```

```
   bp->show();

   return 0;

}
```

If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

Polymorphism

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

```
#include <iostream>

using namespace std;


class Shape {

  protected:

    int width, height;


  public:

    Shape( int a = 0, int b = 0){

      width = a;

      height = b;

    }
    int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};

class Triangle: public Shape {
  public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};
```

```
// Main function for the program

int main() {

   Shape *shape;

   Rectangle rec(10,7);

   Triangle  tri(10,5);


   // store the address of Rectangle

   shape = &rec;


   // call rectangle area.

   shape->area();


   // store the address of Triangle

   shape = &tri;


   // call triangle area.

   shape->area();


   return 0;

}
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage -** the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this

```
class Shape {

  protected:

    int width, height;


  public:

    Shape( int a = 0, int b = 0) {

      width = a;

      height = b;

    }

    virtual int area() {

      cout << "Parent class area :" <<endl;

      return 0;

    }

};
```