UNIT - 5

INHERITANCE
POLYMORPHISM

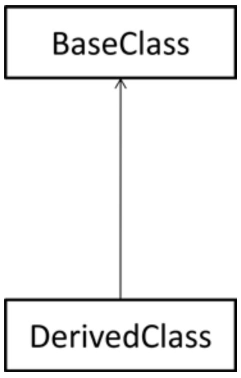
INHERITANCE

Introduction

One of the main advantages of object oriented programming is reusability. Using already existing code is known as reusability. C++ supports reusability through inheritance which is creating a new class from already existing class.

Inheritance is defined as deriving properties and behavior from one class to another class. The existing class from which properties and behavior are derived is known as a base class or super class. The new class into which the properties and behavior are derived is known as a derived class or sub class.

Inheritance is also known as *is-a* relationship between classes. For example a *Employee* is a *Person* and similarly *Programmer* is a *Person*. Common properties and behavior of Employee and Programmer will be available in Person class and specific properties and behavior are present in Employee and Programmer class. Inheritance or is-a relationship between a base class and derived class is represented as follows:



Defining a derived class

Syntax for creating a derived class is as follows:

```
class DerivedClass : [access-specifier] BaseClass
{
	//body of derived class
	...
};
```

In the above syntax *access-specifier* is optional and the valid values for it are *private, protected*, or *public*. If access specifier is not provided, by default it is *private* mode.

Access Specifiers

In C++ there are three access specifiers: *private*, *protected*, and *public*. Let's look at each one of them in detail.

private:
Private is the most restrictive access specifier. Class members which are declared as *private* cannot be accessed outside the class (except in friend classes and friend functions). In private derivation, *protected* and *public* members become *private* members in the derived class and cannot be inherited further.

protected:
Protected is less restrictive than private. Class members which are declared as *protected* are accessible only within the class and in the direct child classes of that class. In protected derivation, protected and public members of base class become protected members of derived class.

public:
Public is the least restrictive access specifier. Class members which declared as *public* are accessible from anywhere. In public derivation, protected and public members of base class become protected and public members of derived class respectively.

Accessibility of the three visibility modes is as shown in the following table:

Access Specifier	**Same Class**	**Derived Class**	**Any Other Class**	**Friend Function**	**Friend Class**
Private	Yes	No	No	Yes	Yes
Protected	Yes	Yes	No	Yes	Yes
Public	Yes	Yes	Yes	Yes	Yes

Following program demonstrates inheritance in C++:

```
#include <iostream>
using namespace std;
class Base
{
	private:
		int x;
	protected:
		int y;
	public:
		int z;
		void show_x()
		{
			cout<<"x = "<<x<<endl;
		}
		void show_y()
		{
			cout<<"y = "<<y<<endl;
		}
```

```
        void show_z()
        {
            cout<<"z = "<<z<<endl;
        }
};
class Derived : public Base
{
    public:
        void display()
        {
            cout<<"This is derived class"<<endl;
        }
};
int main()
{
    Derived d;
    d.display();
    return 0;
}
```

Output of the above program is as follows:

```
This is derived class
```

In the above program variables y and z along with the three public functions are accessible in the *Derived* class. In main class, only z of *Base* class is accessible directly.

Constructors and Destructors in Inheritance

In inheritance following points should be remembered while working with constructors and destructors:

- When a base class contains a constructor, derived class must also have a constructor.
- When a base class constructor contains one or more parameters, the derived class constructor have to pass one or more constructors to the base class.
- When an object for derived class is created, base class constructors is executed first and then derived class constructor gets executed.
- The execution of destructors is in reverse order of constructors execution. Derived class destructor executes first and then base class destructor gets executed.

Following example demonstrates constructors and destructors in inheritance:

```
#include <iostream>
using namespace std;
class Base
{
    private:
        int x;
    protected:
        int y;
    public:
        int z;
        Base()
        {
                cout<<"Base class constructor is called"<<endl;
        }
        void show_x()
        {
                cout<<"x = "<<x<<endl;
        }
        void show_y()
        {
                cout<<"y = "<<y<<endl;
        }
        void show_z()
        {
                cout<<"z = "<<z<<endl;
        }
        ~Base()
        {
                cout<<"Base class destructor is called"<<endl;
        }
};
class Derived : public Base
{
    public:
        Derived()
        {
                cout<<"Derived class constructor is called"<<endl;
        }
        void display()
        {
                cout<<"This is derived class"<<endl;
        }
        ~Derived()
        {
                cout<<"Derived class destructor is called"<<endl;
        }
};
int main()
{
    Derived d;
    d.display();
    return 0;
}
```

Output of the above program is as follows:

```
Base class constructor is called
Derived class constructor is called
```

```
This is derived class
Derived class destructor is called
Base class destructor is called
```

Invoking Base Class Constructor with Parameters

When a base class constructor contains one or more parameters, it is the responsibility of derived class constructor to pass the required parameters to the base class constructor. Syntax for passing parameters to base class constructor is as follows:

```
DerivedClass(params-list) : BaseClas(params-list)
{
	//body of derived class constructor
	...
}
```

Following program demonstrates invoking base class constructor with parameters from a derived class constructor:

```
#include <iostream>
using namespace std;
class Base
{
	private:
		int x;
	protected:
		int y;
	public:
		int z;
		Base(int x, int y, int z)
		{
			this->x = x;
			this->y = y;
			this->z = z;
		}
		void show_x()
		{
			cout<<"x = "<<x<<endl;
		}
		void show_y()
		{
			cout<<"y = "<<y<<endl;
		}
		void show_z()
		{
			cout<<"z = "<<z<<endl;
		}
};
class Derived : public Base
```

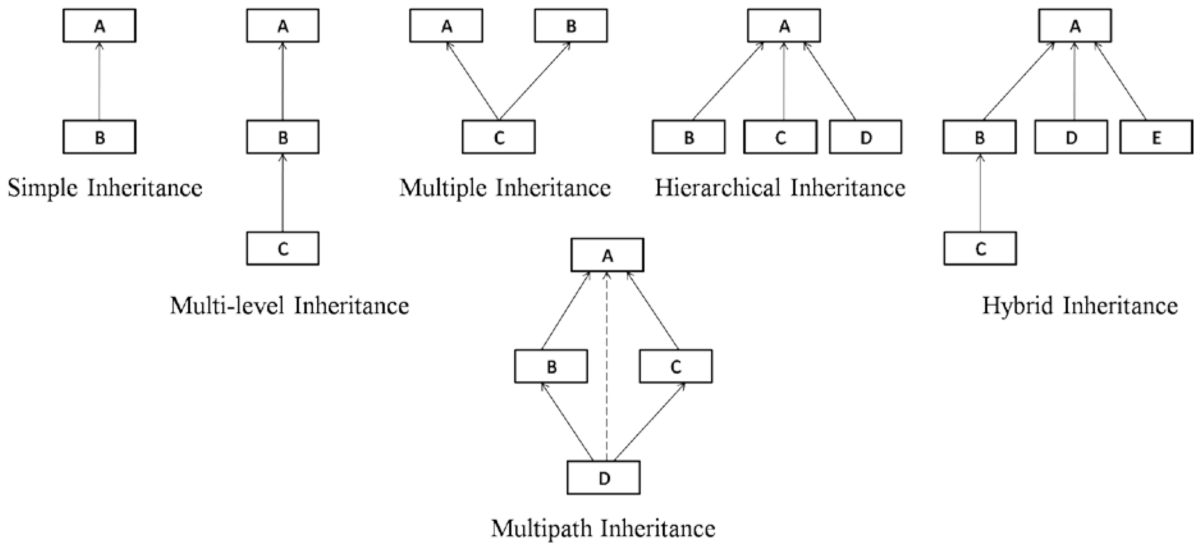
```
{
        public:
                Derived(int x, int y, int z) : Base(x, y, z){ }
                void display()
                {
                        cout<<"This is derived class"<<endl;
                }
};
int main()
{
        Derived d(10, 20, 30);
        d.show_x();
        d.show_y();
        d.show_z();
        d.display();
        return 0;
}
```

Output of the above program is as follows:

```
x = 10
y = 20
z = 30
This is derived class
```

Types of Inheritance

Following figure illustrates different types of inheritance in C++:



Simple Inheritance

In simple inheritance, there is only one super class and one sub class. This is the simplest of all the types of inheritance. Following program demonstrates simple inheritance:

```
#include<iostream>
using namespace std;
class A
{
	public:
		A()
		{
			cout<<"This is super class A"<<endl;
		}
};
class B : public A
{
	public:
		B()
		{
			cout<<"This is sub class B"<<endl;
		}
};
int main()
{
	B obj;
}
```

Output for the above program is a follows:

```
This is super class A
This is sub class B
```

Multi-Level Inheritance

In multi-level inheritance, a sub class inherits a super class and once again that sub class is inherited by another sub class and so on. At each level we have one super class and one sub class. Following program demonstrates multi-level inheritance:

```
#include<iostream>
using namespace std;
class A
{
	public:
		A()
		{
			cout<<"This is super class A"<<endl;
		}
};
class B : public A
{
	public:
		B()
		{
			cout<<"This is sub class B of super class A"<<endl;
		}
};
class C : public B
{
	public:
		C()
		{
			cout<<"This is sub class C of super class B"<<endl;
		}
};
int main()
{
	C obj;
}
```

Output for the above program is as follows:

```
This is super class A
This is sub class B of super class A
This is sub class C of super class B
```

Multiple Inheritance

In multiple inheritance, multiple super classes will be inherited by a single sub class. In multiple inheritance there might be ambiguity problem in sub class. For example if super class A contains a variable *x* and another super class B also contains a variable *x*, then in sub class C, when *x* is referred, compiler will be confused whether it belongs to class A or class B. To avoid this ambiguity, we use the scope resolution operator.

Following program demonstrates multiple inheritance and the solution to ambiguity problem:

```
#include<iostream>
using namespace std;
class A
{
	public:
		int x;
		A(int x)
		{
			this->x = x;
		}
		void show()
		{
			cout<<"A's x = "<<x<<endl;
		}
};
class B
{
        public:
                int x;
                B(int x)
                {
                        this->x = x;
                }
                void show()
                {
                        cout<<"B's x = "<<x<<endl;
                }
};
class C : public A, public B  //multiple inheritance
{
        public:
                C(int a, int b) : A(a), B(b)
                {
                        A::show();
                        B::show();
                }
};
int main()
{
        C objC(10, 20);
}
```

Output for the above program is as follows:

```
A's x = 10
B's x = 20
```

Hierarchical Inheritance

In hierarchical inheritance, a single super class is inherited by multiple sub classes. Following program demonstrates hierarchical inheritance:

```
#include<iostream>
using namespace std;
class A
{
        public:
                A()
                {
                        cout<<"Super class A's constructor"<<endl;
                }
};
class B : public A
```

```
{
        public:
                B()
                {
                        cout<<"Sub class B's constructor"<<endl;
                }
};
class C : public A
{
        public:
                C()
                {
                        cout<<"Sub class C's constructor"<<endl;
                }
};
int main()
{
        C objC;
}
```

Output for the above program is as follows:

```
Super class A's constructor
Sub class C's constructor
```

Hybrid Inheritance

This is a combination of any two or more of the types of inheritance mentioned before. Following program demonstrates hybrid inheritance:

```
#include<iostream>
using namespace std;
class A
{
        public:
                A()
                {
                        cout<<"Super class A's constructor"<<endl;
                }
};
class B : public A
{
        public:
                B()
                {
                        cout<<"Sub class B's constructor"<<endl;
                }
};
class C : public A
```

```
{
	public:
		C()
		{
			cout<<"Sub class C's constructor"<<endl;
		}
};
class D: public B
{
	public:
		D()
		{
			cout<<"Sub class D's constructor"<<endl;
		}
};
int main()
{
	D objD;
}
```

Output for the above program is as follows:

```
Super class A's constructor
Sub class B's constructor
Sub class D's constructor
```

Multipath Inheritance

Multipath inheritance is a special case of hybrid inheritance. It is a combination of multiple, hierarchical, and multi-level inheritance.

Diamond Problem

In multipath we encounter an ambiguous situation which is also known as diamond problem. For example, let the super class A contains a variable x, which is inherited by sub classes B and C. Now, sub class D of super classes B and C will be able to access the variable x. So, whose x is it? Is it B's x or C's x? To avoid this ambiguous situation and to maintain only one copy of x, we mark the base class A as virtual class. Syntax for making A as virtual base class is as follows:

```
class B : virtual public A { ... }
class C : virtual public A { ... }
```

Following program demonstrates multipath inheritance and virtual base class solution for the diamond problem:

```
#include<iostream>
using namespace std;
class A
{
        protected:
                int x;
        public:
                A()
                {
                        cout<<"Super class A's constructor"<<endl;
                }
                void read()
                {
                        cout<<"Enter value of x: ";
                        cin>>x;
                }
                void show()
                {
                        cout<<"x = "<<x;
                }
};
class B : virtual public A
{
        public:
                B()
                {
                        cout<<"Sub class B's constructor"<<endl;
                }
};
class C : virtual public A
{
        public:
                C()
                {
                        cout<<"Sub class C's constructor"<<endl;
                }
};
class D: public B, public C
{
        public:
                D()
                {
                        cout<<"Sub class D's constructor"<<endl;
                }
};
int main()
{
        D objD;
        objD.read();
        objD.show();
        return 0;
}
```

Output for the above program is as follows:

```
Super class A's constructor
Sub class B's constructor
Sub class C's constructor
Sub class D's constructor
Enter value of x: 20
x = 20
```

Note: The constructor of a virtual base class is invoked before any non-virtual base class. If there are multiple virtual base classes, then they are invoked in the order in which they are declared.

Object Slicing

In inheritance we can assign a derived class object to a base class object. But, a base class object cannot be assigned to a derived class object. When a derived class object is assigned to a base class object, extra features provided by the derived class will not be available. Such phenomenon is called object slicing.

Following program demonstrates object slicing:

```
#include<iostream>
using namespace std;
class A
{
	protected:
		int x;
	public:
		void show()
		{
			cout<<"x = "<<x<<endl;
		}
};
class B : public A
{
	protected:
		int y;
	public:
		B(int x, int y)
		{
			this->x = x;
			this->y = y;
		}
		void show()
		{
			cout<<"x = "<<x<<endl;
			cout<<"y = "<<y<<endl;
		}
};
int main()
{
        A objA;
        B objB(30, 20);
        objA = objB;   //Assign derived class object to base class object
        objA.show();
        return 0;
}
```

Output of the above program is as follows:

```
x = 30
```

In the above program, when objA.show() is called, even though class B has variable *y*, we were not able to access it; which is treated as object slicing.

Pointer to Derived Class

A base class pointer can point to a derived class object. But, a derived class pointer can never point to a base class object. Following program demonstrates object slicing using a base class pointer:

```
#include<iostream>
using namespace std;
class A
{
        protected:
                int x;
        public:
                void show()
                {
                        cout<<"x = "<<x<<endl;
                }
};
class B : public A
{
        protected:
                int y;
        public:
                B(int x, int y)
                {
                        this->x = x;
                        this->y = y;
                }
                void show()
                {
                        cout<<"x = "<<x<<endl;
                        cout<<"y = "<<y<<endl;
```

```
                    }
};
int main()
{
          A *bptr;
          B objB(30, 20);
          bptr = &objB; //Assign derived class object to base class pointer
          bptr->show();
          return 0;
}
```

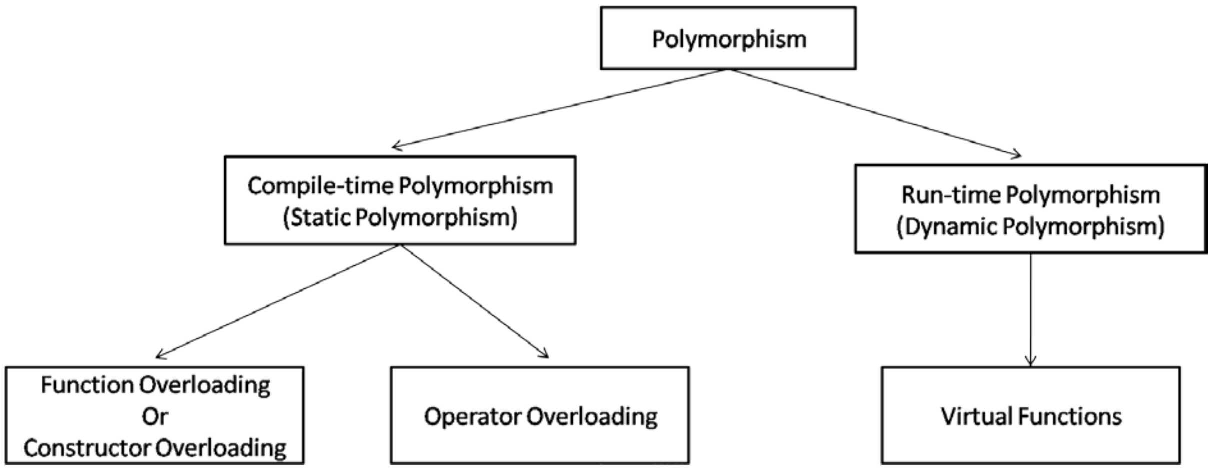
Output of the above program is as follows:

```
x = 30
```

POLYMORPHISM

Introduction

Polymorphism is one of the key features of object orientation. It means many forms or single interface multiple implementations. Polymorphism is of two types: 1) Compile-time polymorphism and 2) Run-time polymorphism as illustrated in the following figure:


Polymorphism
Compile-time Polymorphism (Static Polymorphism)
Run-time Polymorphism (Dynamic Polymorphism)
Function Overloading Or Constructor Overloading
Operator Overloading
Virtual Functions

Function Overriding

When a base class and sub class contains a function with the same signature, and the function is called with base class object, then the function in derived class executes and the function in base class is said to be overridden. This is known as function overriding. Following program demonstrates function overriding:

```
#include<iostream>
using namespace std;
class A
{
	protected:
		int x;
	public:
		void show()
		{
			cout<<"x = "<<x<<endl;
		}
};
class B : public A
{
	protected:
		int y;
	public:
		B(int x, int y)
		{
			this->x = x;
			this->y = y;
		}
		void show()
		{
			cout<<"x = "<<x<<endl;
			cout<<"y = "<<y<<endl;
		}
};
int main()
{
	A objA;
	B objB(30, 20);
	objB.show();
	return 0;
}
```

Output for the above program is:

```
x = 30
y = 20
```

In the above program, both super class A and sub class B contains the same function *show()* with same signature (function name plus parameters). So, when sub class object is used to call *show()*, function in sub class B executes overriding the function in super class A.

Note: Overloading of functions is not supported across classes (in inheritance) in C++.

Virtual Functions

We know that when a base class pointer refers to a derived class object, the extra features in derived class are not available. To access the extra features in the derived class, we make the functions in the base class as virtual. Syntax for creating a virtual function is as follows:

```
virtual return-type function-name(params-list)
{
	//Body of function
	...
}
```

A class which contains one or more virtual functions is known as a polymorphic class. Following program demonstrates accessing derived class features using virtual functions:

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

```
{
	protected:
		int x;
	public:
		virtual void show()	//virtual function
		{
			cout<<"x = "<<x<<endl;
		}
};
class B : public A
{
	protected:
		int y;
	public:
		B(int x, int y)
		{
			this->x = x;
			this->y = y;
		}
		void show()
		{
			cout<<"x = "<<x<<endl;
			cout<<"y = "<<y<<endl;
		}
};
int main()
{
	A *bptr;
	B objB(30, 20);
	bptr = &objB;
	bptr->show();
	return 0;
}
```

Output of the above program is as follows:

```
x = 30
y = 20
```

Rules for Virtual Functions

Following points must be remembered while working with virtual functions:

- Virtual functions must be members of a class.
- Virtual functions must be created in public section so that objects can access them.
- When virtual function is defined outside the class, virtual keyword is required only in the function declaration. Not necessary in the function definition.
- Virtual functions cannot be static members.
- Virtual functions must be accessed using a pointer to the object.

- A virtual function cannot be declared as a friend of another class.
- Virtual functions must be defined in the base class even though it does not have any significance.
- The signature of virtual function in base class and derived class must be same.
- A class must have a virtual destructor but it cannot have a virtual constructor.

Pure Virtual Functions

When the code for virtual function in a base class is insignificant, we can make such virtual functions as pure virtual functions. A pure virtual function is a virtual function without any definition. Syntax for creating a pure virtual functions is as follows:

virtual return-type function-name(params-list) = 0;

A class which contains at least one pure virtual function is called a abstract class and the class which provides the definition for the pure virtual function is called a concrete class.

Late Binding (Dynamic Polymorphism)

In inheritance when a derived class object is assigned to a base class pointer, and a polymorphic function is invoked, the function call is linked to the function definition at run-time. Such postponement of linkage to run-time instead of compile-time is called as late binding or dynamic polymorphism.

Abstract Class

A class which contains at least one pure virtual function is called an abstract class. Since abstract class is incomplete, another class should derive it and provide definitions for the pure virtual functions in the abstract class.

Following are the features of an abstract class:

- Abstract class must contain at least one pure virtual function.
- Objects cannot be created for an abstract class. But pointers can be created.
- Classes inheriting an abstract class must provide definitions for all pure virtual functions in the abstract class. Otherwise, the sub class also becomes an abstract class.
- An abstract class can have non-virtual functions and data members.

Following are the uses of an abstract class:

- Abstract class provides a common standard interface for all the sub classes.
- Abstract classes allow new features to be easily added to an existing application.

Following program demonstrates pure virtual functions, late binding and an abstract class:

```
#include <iostream>
using namespace std;
class Shape     //Abstract class
{
        public:
                virtual void area() = 0;
                virtual void peri() = 0;
};
class Rectangle : public Shape          //Concrete class
{
        private:
                int l;
                int b;
        public:
                Rectangle(int l, int b)
                {
                        this->l = l;
                        this->b = b;
                }
                void area()
                {
                        cout<<"Area of rectangle: "<<(l*b)<<endl;
                }
                void peri()
                {
                        cout<<"Perimeter of rectangle: "<<2*(l+b)<<endl;
                }
};
class Circle : public Shape     //Concrete class
{
        private:
                int r;
        public:
                Circle(int r)
                {
                        this->r = r;
                }
                void area()
                {
                        cout<<"Area of circle: "<<(3.14*r*r)<<endl;
                }
                void peri()
                {
                        cout<<"Perimeter of circle: "<<(2*3.14*r)<<endl;
                }
};
int main()
{
        Shape *s;
        Rectangle r(10, 20);
        Circle c(4);
        s = &r;
        s->area();      //late binding
        s->peri();      //late binding
        s = &c;
        s->area();      //late binding
        s->peri();      //late binding
        return 0;
}
```

Output for the above program is as follows:

```
Area of rectangle: 200
Perimeter of rectangle: 60
Area of circle: 50.24
Perimeter of circle: 25.12
```

Virtual Constructors and Destructors

C++ allows programmers to create virtual destructors. But, it doesn't allow virtual constructors to be created because of various reasons. To know why a virtual destructor is needed, consider the following program:

```
#include <iostream>
using namespace std;
class A
{
        public:
                A()
                {
                        cout<<"A's constructor"<<endl;
                }
                ~A()
                {
                        cout<<"A's destructor"<<endl;
                }
};
class B : public A
{
        public:
        B()
        {
                cout<<"B's constructor"<<endl;
        }
        ~B()
        {
                cout<<"B's destructor"<<endl;
        }
};
int main()
{
        A *bptr = new B();
        delete bptr;
        return 0;
}
```

Output for the above program is as follows:

```
A's constructor
B's constructor
A's destructor
```

From the above output you can see that derived class destructor didn't execute. This might lead to problems like memory leakage. To avoid such problems we make destructor in base class as virtual destructor.

Virtual destructor instructor ensures that the derived class destructor is executed. To create a virtual destructor, precede the destructor definition with *virtual* keyword. Following program demonstrates a virtual destructor:

```
#include <iostream>
using namespace std;
class A
{
        public:
                A()
                {
                        cout<<"A's constructor"<<endl;
                }
                virtual ~A()
                {
                        cout<<"A's destructor"<<endl;
                }
};
class B : public A
{
        public:
        B()
        {
                cout<<"B's constructor"<<endl;
        }
        ~B()
        {
                cout<<"B's destructor"<<endl;
        }
};
int main()
```

```
{
        A *bptr = new B();
        delete bptr;
        return 0;
}
```

Output for the above program is as follows:

```
A's constructor
B's constructor
B's destructor
A's destructor
```

Now you can see that derived class constructor is also executed.

Advantages and Disadvantages of Inheritance

Advantages of inheritance are as follows:

- Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inherited class.
- Reusability enhanced reliability. The base class code will be already tested and debugged.
- As the existing code is reused, it leads to less development and maintenance costs.
- Inheritance makes the sub classes follow a standard interface.
- Inheritance helps to reduce code redundancy and supports code extensibility.
- Inheritance facilitates creation of class libraries.

Disadvantages of inheritance are as follows:

- Inherited functions work slower than normal function as there is indirection.
- Improper use of inheritance may lead to wrong solutions.
- Often, data members in the base class are left unused which may lead to memory wastage.
- Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.