Computer Programing [Lab]

Lab # 08: Inheritance & Composition

Agenda

- Inheritance
- Public, Private & Protected Inheritance
- Types of Inheritance
- Composition
- Function Overriding
- Issues in Inheritance

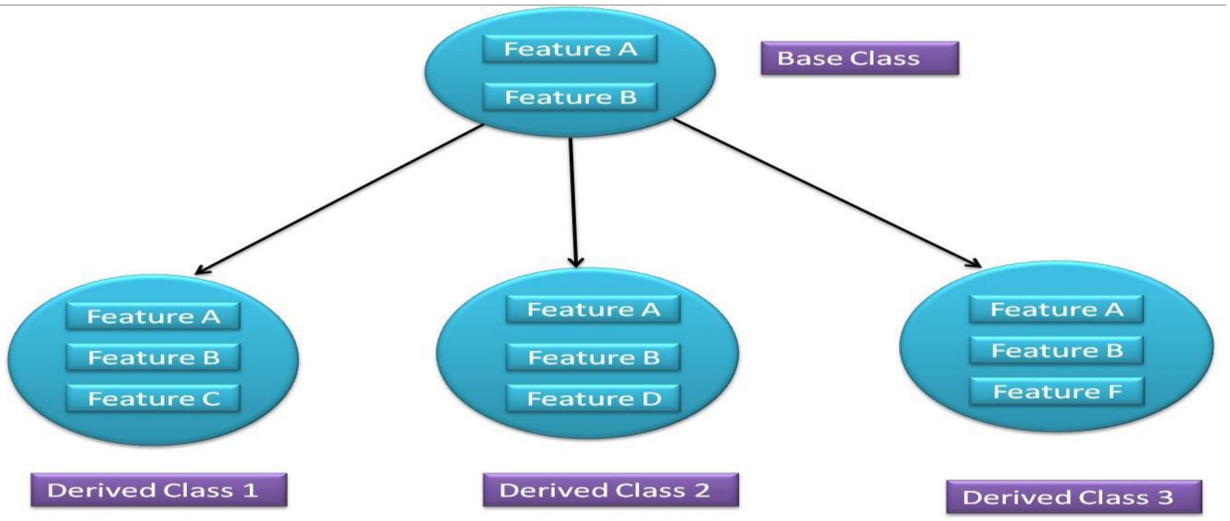
Inheritance

A C++ strongly supports the concept of **Reusability**. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called inheritance is.

The old class is referred to as the base class and the new one is called the derived class or subclass. A derived class includes all features of the generic base class and then adds qualities specific to the derived class.

Inheritance is the process by which objects of one class acquire the properties of objects of another class in the hierarchy. Once a super class is written and debugged, it need not be touched again but at the same time can be adapted to work in different situations. Reusing existing code saves time and money and increases a program's reliability.

For example, the scooter is a type of the class two-wheelers, which is again a type of (or kind of) the class motor vehicles. As shown in the below diagram the principle behind it is that the derived class shares common characteristics with the class from which it is derived.



New classes can be built from the existing classes. It means that we can add additional features to an existing class without modifying it. The new class is referred as derived class or subclass and the original class is known as base classes or super class.

Inheritance is either public, protected, or private

1. Public Inheritance

If Member Access Specifier is public that is, the inheritance is public then:

a. The public members of A are public members of B. They can be directly accessed in class B.
b. The protected members of A are protected members of B. They can be directly accessed by the member functions (and friend functions) of B.
c. The private members of A are hidden in B. They cannot be directly accessed in B. They can be accessed by the member functions (and friend functions) of B through the public or protected members of A.

Example

```
//Sample program for understanding public inheritance

#include <iostream>
using namespace std;
class a
{
	public:
		int d;
		void fun1()
			{ e=2; }

	private: int e;
	protected: int c;
};
class b:public a
{
	public:
		int f;
		void fun()
		{
			d=2; c=5; // Protected member can be accessed here e=5 ;
			          // error private can't be accessed here fun1();
		}
};
int main()
{
	b obj;
	obj.d=3; //public member directly accessed here g.c=4;
	          //protected member can't be accessed here g.fun();
	system("pause");
}
```

2. Protected Inheritance

If Member Access Specifier is protected that is, the inheritance is protected then:

a. The public members of A are protected members of B. They can be accessed by the member functions (and friend functions) of B.
b. The protected members of A are protected members of B. They can be accessed by the member functions (and friend functions) of B.
c. The private members of A are hidden in B. They cannot be directly accessed in B. They can be accessed by the member functions (and friend functions) of B through the public or protected members of A.

Example

```
// This program demonstrates the use of classes and objects
#include <iostream>
using namespace std;
class a
{
	public:
		int d;
		void fun1()
		{ e=2; }

	private: int e;
	protected: int c;
};
class b:protected a
{
	public:
		int f;
		void fun()
		{
			d=2;
			c=5; // Protected member can be accessed here
			e=5 ; // error private can't be accessed here
			fun1();
		}
};
int main()
{
	b obj;
	obj.d=3;  //public member can’t be accessed here directly
	         //because now it is protected member
	obj.c=4;//protected member can't be accessed here
	system("pause");
}
```

3. Private Inheritance

If Member Access Specifier is private that is, the inheritance is private then:

a. The public members of A are private members of B. They can be accessed by the member functions (and friend functions) of B.
b. The protected members of A are private members of B. They can be accessed by the member functions (and friend functions) of B.
c. The private members of A are hidden in B. They cannot be directly accessed in B. They can be accessed by the member functions (and friend functions) of B through the public or protected members of A.

Example

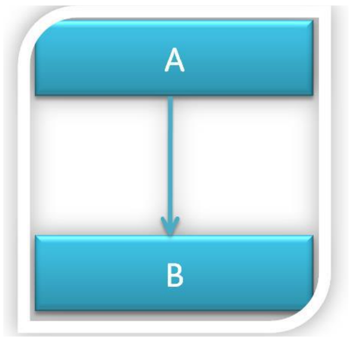
```
// This program demonstrates the use of constructors
#include <iostream>
using namespace std;
class a
{
	public:
		int d;
		void fun1()
		{ e=2; }

	private: int e;
	protected: int c;
};
class b:private a
{
	public:
		int f;
		void fun()
		{
		d=2; // public can be accessed because now this is private member
		c=5; // Protected member can be accessed here because now this is private member
		}
};
int main()
{
    b obj;
    obj.d=3; // error public member can't be accessed here directly
            //because now it is private member
    obj.fun();//protected member can't be accessed here
    system("pause");
}
```

Types of Inheritance

1. Single Level Inheritance
2. Multiple Inheritance
3. Hierarchical inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

Single Level Inheritance

A derived class with only one base class is called single inheritance. Consider a simple example of single inheritance. In this program show a base class A and derived class B. The class A contains one private data member, one public data member, and three public member functions. The class B contains one private data members and two public member functions.

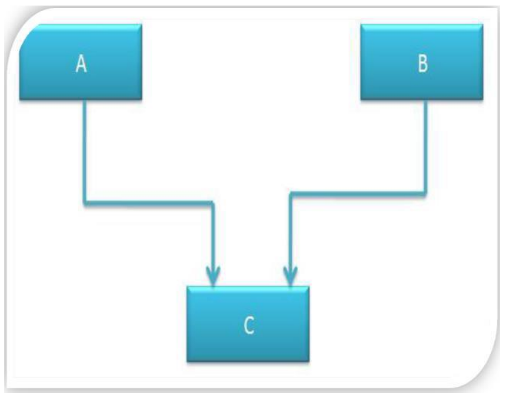


Example

```
// Example: Single Level Inheritance
#include<iostream>
using namespace std;
class A
{
       int a;
       public:
              int b;
              void get_ab();
              int get_a();
              void show_a();
};
class B: public A {
       int c;
       public:
              void mul();
              void display();
};
void A::get_ab()
       { a=5; b=10; }
int A:: get_a()
       { return a;}
void A:: show_a()
       { cout<< "a="<<a<< "\n" ;}
void B:: mul()
       { c=b*get_a();}
void B::display()
{
       cout<< "a="<<get_a();       cout<< "b="<<b;
       cout<< "c="<<c;
}

int main()
{
       B d;
       d.get_ab();
       d.mul();
       d.show_a();
       d.display();
       d.b=20;
       d.mul();
       d.display();
       return 0;
}
```

Multiple Inheritance

A class can inherit properties from more than one class which is known as multiple inheritances. This form of inheritance can have several super classes. A class can inherit the attributes of two or more classes as shown below diagram. Multiple inheritances allow us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligent if another.

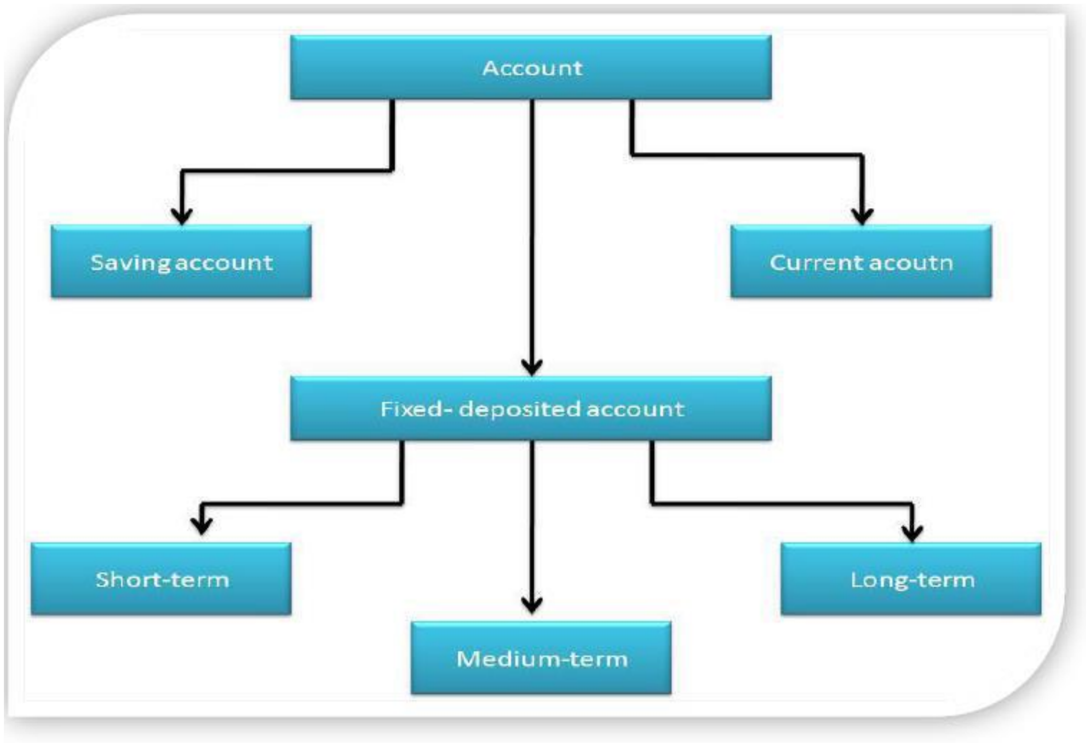


Example

```
// Example: Multiple Inheritance
#include<iostream>
using namespace std;
class M
{
        protected:
                int m;
        public :
                void get_m(int);
};
class N
{
        protected:
                int n;
        public :
                void get_n(int);
};

class P: public M, public N
{
        public :
                void display();
};
void M :: get_m(int x) { m=x; }
void N::get_n(int y) { n=y; }
void P:: display() { cout<<"m="<<m<<"\n"<<"n="<<n<<"\n"<<"m*n="<<m*n<<"\n"; }

int main()
{
        P p1;
        p1.get_m(10);
        p1.get_n(20);
        p1.display();
        return 0;
}
```

Hierarchical Inheritance

When the properties of one class are inherited by more than one class, it is called hierarchical inheritance. This form has one super class and many Subclasses. More than one class inherits the traits of one class. For example: bank accounts.



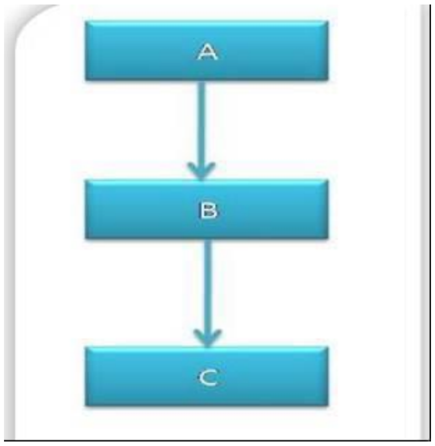
Example

```
// Example: Multiple Inheritance
#include<iostream>
using namespace std;
class M
{
        protected:
                int m;
        public :
                void get_m(int);
};
class N: public M
{
        protected:
                int n;
        public :
                void get_n(int);
};
class P: public M
{
        public :
                void display();
};
```

```
void M :: get_m(int x) { m=x; }
void N::get_n(int y) { n=y; }
void P:: display() { cout<<"m="<<m<<"\n";
}

int main()
{
        P p1;
        N n1;
        p1.get_m(10);
        n1.get_m(15);
        p1.display();
        return 0;
}
```

Multi-Level Inheritance

A class can be derived from another derived class which is known as multilevel inheritance. Order of Constructor Calling in Multilevel Inheritance, when the object of a subclass is created the constructor of the subclass is called which in turn calls constructor of its immediate super class. For example, if we take a case of multilevel inheritance, where class B inherits from class A, and class C inherits from class B, which show the order of constructor calling.



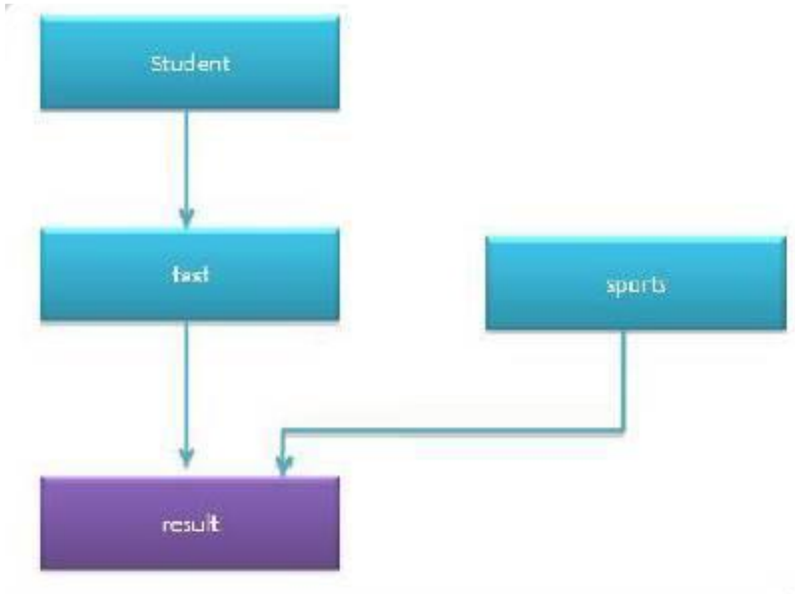
Example

```
// Example: Multi Level Inheritance
#include<iostream>
using namespace std;
class M
{
        protected:
                int m;
        public :
                void get_m(int);
};
class N: public M
{
        protected:
                int n;
        public :
                void get_n(int);
};

class P: public N
{
        public :
                void display();
```

```
void M :: get_m(int x) { m=x; }
void N::get_n(int y) { n=y; }
void P:: display() { cout<<"m="<<m<<"\n";
}

int main()
{
        P p1;
        N n1;
        n1.get_m(10);
        p1.get_n(15);
        p1.display();
        return 0;
}
```

Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design one inheritance called hybrid inheritance. For instance, consider the case of processing the student results, the weightage for sport is stored in separate classes.



Example

```
// Example Hybrid Inheritance
#include <iostream>
using namespace std;
class stud
{
       protected:    int rno;
       public:
              void getno(int n)
                     { rno=n; }
              void display_rno()
                     { cout<<"Roll_no="<<rno<<"\n"; }
};
class test: public stud
{
       protected: int sub1,sub2;
       public: void get_mark(int m1,int m2)
                     { sub1=m1; sub2=m2; }
               void display_mark()
              {
                cout<<"sub1"<<sub1<<"\n";
                cout<<"sub2"<<sub2<<"\n";
              }
};
class sports
{
       protected: float score;
       public :
       void get_score(float s)
              { score=s; }
       void put_score()
              { cout<<"Sort :"<<score<<"\n"; }
};
class result: public test ,public sports
{
       float total;
       public:
              void display(); };
              void result::display()
              {
                     total=sub1+sub2+score;
                     display_rno();
                     display_mark();
                     put_score();
                     cout<<" total score: "<<total<<"\n";
              }
int main()
{
       result r1;
       r1. getno(123);
       r1. get_mark(60,80);
       r1.get_score(6);
       r1.display();
}
```

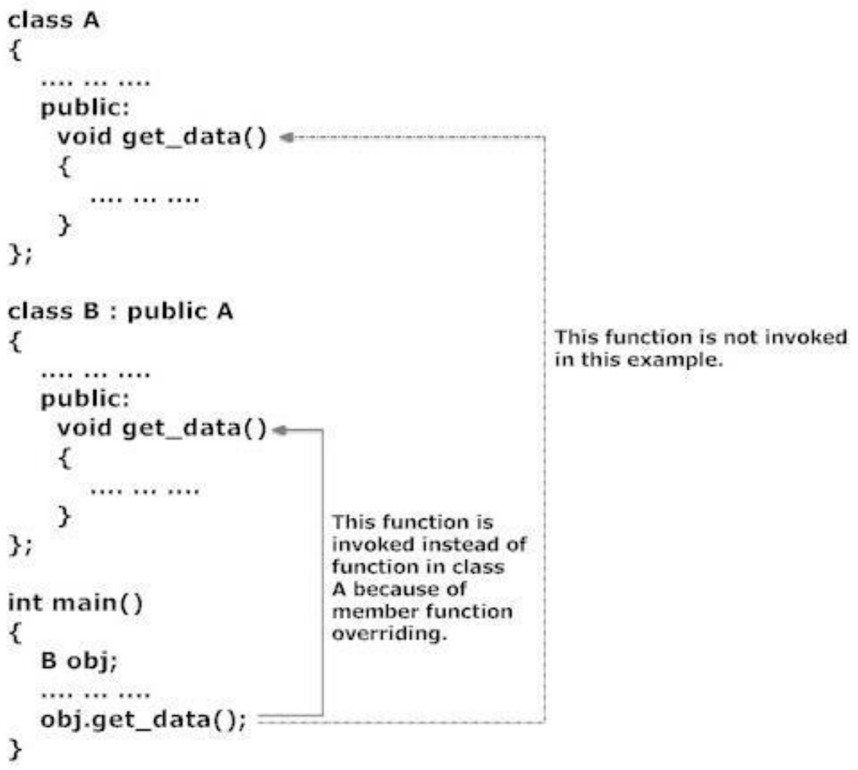
Composition (Aggregation)

Composition (aggregation) is another way to relate two classes. In composition (aggregation), one or more members of a class are objects of another class type.

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

class Bar
{
        public: int baz;
};
class Foo
{
        public:
        Bar bar;
        Foo(int x) {  bar.baz=x; }
};
```

C++ Function Overriding

If base class and derived class have member functions with same name and arguments. If you create an object of derived class and write code to access that member function then, the member function in derived class is only invoked, i.e., the member function of derived class overrides the member function of base class. This feature in C++ programming is known as function overriding.



```
class A
{
   .... ... ....
   public:
    void get_data()
    {
       .... ... ....
    }
};

class B : public A
{
   .... ... ....
   public:
    void get_data()
    {
       .... ... ....
    }
};

int main()
{
   B obj;
   .... ... ....
   obj.get_data();
}
```

This function is not invoked in this example.

This function is invoked instead of function in class A because of member function overriding.

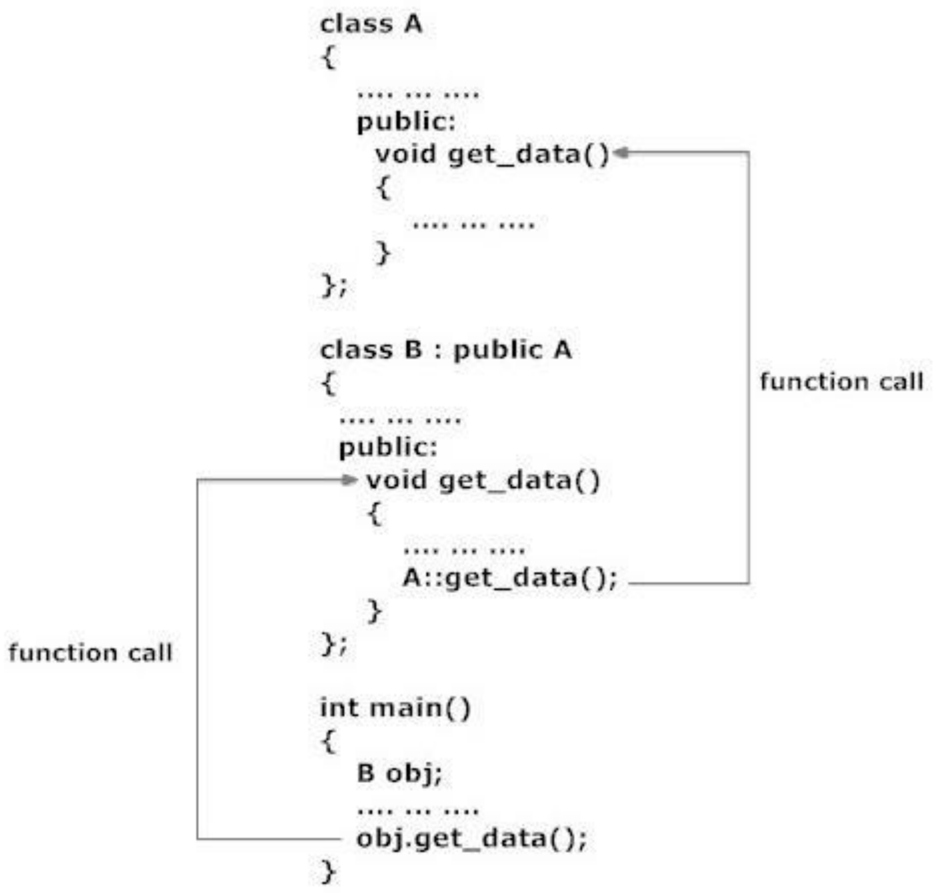
Figure: Member Function Overriding in C++

Accessing the Overridden Function in Base Class From Derived Class

To access the overridden function of base class from derived class, scope resolution operator::. For example: If you want to access get_data() function of base class from derived class in above example then, the following statement is used in derived class.

```
A::get_data; // Calling get_data() of class A.
```

It is because, if the name of class is not specified, the compiler thinks get_data()function is calling itself.



```
class A
{
   .... ... ....
   public:
     void get_data()
     {
        .... ... ....
     }
};

class B : public A
{
  .... ... ....
  public:
    void get_data()
    {
       .... ... ....
       A::get_data();
    }
};

int main()
{
   B obj;
   .... ... ....
   obj.get_data();
}
```

Issues in Inheritance

1. Function overriding
2. How to calling base class constructor with parameter