INHERITANCE:

EXTENDING CLASSES

INTRODUCTION:

Reusability is yet another important feature of OOP. If we could reuse something that already exists rather than trying to create the same all over again. It reduce frustration and increase reliability. Fortunately, C++ strongly supports the concept of reusability. 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 is called inheritance. The old class is referred to as the base class and the new one is called the derived class or subclass.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level.

- A derived class with only one base class is called single inheritance.
- One with several classes is called multiple inheritance.
- The traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance.
- The mechanism of deriving a class from another 'derived class' is known as multiple inheritance.

DEFINING DERIVED CLASSES:

The general form of defining a derived class is:

```
Class derived-class-name: visibility-mode base-class-name
{
… //
… // member of derived class
…//
};
```

The colon indicates that the derived-class-name is derived from then base-class-name. The visibility-mode is optional. It may be either private or public. The default visibility-mode is private. Its base class are privately or publicly derived.

Ex:

```
Class derived: private base                    // private derivation
{
```

```
members of derived
};
Class derived: public base                    // public derivation
{
members of derived
};
Class derived: base                    // private derivation by default
{
members of derived
};
```

When a base class is privately inherited by a derived class, 'public members' of the base class becomes 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. A public member of a class can be accessed by its own objects using dot operator. When the base class is publicly inherited, 'public member' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. In both the cases, the private members are not inherited, the private members of a base class will never become the members of its derived class.

Inheritance, some of the base class data elements and member functions 'inherited' into the derived class. We can add data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerfull tool for incremental program development.

SINGLE INHERITANCE:

A derived class with only one base class is called single inheritance.

Base class B and derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.:

PROGRAM:

```
#include <iostream>
using amespace std;
class B
{
int a;                          // private; not inheritable
public:
int b;                          // private; ready for inheritance
void set_ab( );
int get_a (void);
void show_a (void);
};
class D : public B              // public derivation
{
int c;
public:
void mul (void);
void display (void);
};
//-------------------------------------------------------------------------
Void B :: set_ab (void)
{
a=5; b=10;
}
int B :: get_a( ) B
{
return a;
```

```
 }
Void B :: show_a( )
{
Cout<< ”a= “ << a << ”\n”;
}
Void B :: mul( )
{
c = b * get_a ( );
}
Void B :: display( )
{
Cout<< ”a= “ << get_a<< ”\n”;
Cout<< ”b= “ <<  b << ”\n”;
Cout<< ”c = “ <<  c << ”\n\n”;
}
//-------------------------------------------------------------------------------------------
int main ( )
{
D d;
d.set_ab ( );
d.mul ( );
d.show_a ( );
d.display ( );
d.b = 20;
d.mul ( );
d.display ( );
return 0;
```

}

OUTPUT:

a = 5

a = 5

b = 10

c = 50

a = 5

b = 20

c = 100

The class D is a public derivation of the base class B. Therefore, D inherits all the public members of B and retains their visibility. Thus, a public member of the base class B is also a public member of the derived class D. The private members of B cannot inherited by D. The class D, in effect, will have more members than what it contains at the time of declaration.

MAKING A PRIVATE MEMBER INHERITABLE:

If the private data needs to be inherited by a derived class. Modifying the visibility limit of the private member by making it public. It accessible to all the other functions of the program, thus taking away the advantage of data hiding.

C++ provides a third visibility modifier, protected, which serve a limited purpose in 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 the functions outside these two classes. A class can now use all three visibility modes.

```
class alpha
{
private:                    // optional
…                           // visible to member functions
…                           //  within its class
protected:
…                           // visible to member functions
…                           // of its own and that of derived class
```

Public:

… // visible to all functions

… // in the program

};

When a protected member is inherited in public mode, it becomes protected in the derived class too and therefore is accessible by the member functions of the derived class. A protected member, inherited in the private mode derivation, become private in the derived class.

The keyword private, protected, and public may appear in any order and any number of times in the declaration of a class.

class beta

{

protected:

…

public:

…

private:

…

public:

…

};

Is a valid class definition.

class beta

{

…

…

protected:

…

public:

…

}

It is also possible to inherit a base class in protected made. In protected derivation, both the public and protected members of the base class become protected members of the derived class.

Visibility of inherited members:

Base class visibility	public Derivation	Derived class visibility	protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

1. A function that is a friend of the class.
2. A member function of a class that is a friend of the class.
3. A member function of a derived class.

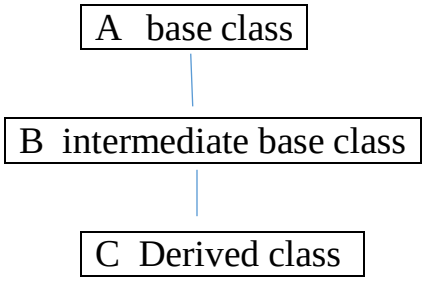
While the friend functions and the member functions of a friend class can have direct access to both the private and protected data, the member functions of a derived class class can directly access only the protected data. However, they can access private data through the member functions of the based class.

MULTILEVEL INHERITANCE:

The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. The class B is known as intermediate base class since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.

A derived class with multilevel inheritance is declared as follows:

```
Class A{…};                    // Base class
Class B: public A {…};         // B derived from A
Class C: public B {…};         // C derived from B
```



This process can be extended to any number of levels.

PROGRAM:

```
#include <iostream>
using namespace std;
class student
{
protected:
int roll_number;
public:
void get_number(int);
void put_number(void);
};
void student :: get_number(int a)
{
roll_number = a;
}
void student :: put_number( )
{
cout<< “Roll number;” << roll_number<<”\n”;
}
class test : public student                    // first level derivation
{
protected:
float sub1;
float sub2;
public:
void get_marks (float,float);
void put_marks(void);
```

```
};
void test :: get_marks(float x, float y)
{
sub1 = x;
sub2 = y;
}
void test :: put_marks( )
{
cout<< “Marks in SUB1= ” << sub1<<”\n”;
cout<< “Marks in SUB2= ” << sub2<<”\n”;
}
Class result : public test                    // second level kderivation
{
Float total;
Public:
Void display(void);
};
Void result :: display(void)
{
total= sub1+sub2;
put_number();
put_marks();
cout<< “Total = ”<< total << “\n”;
}
int main ( )
{
result student1;                              // student1 created
```

```
student1.get_number(111);
student1.get_marks(75.0,59.5);
student1.diaplay( );
return 0
}
```

OUTPUT:

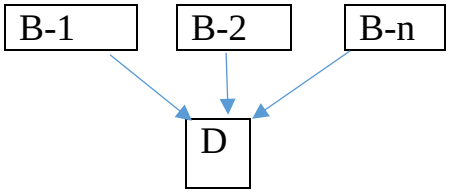
Roll number: 111

Marks in SUB1 = 75

Marks in SUB2 = 59.5

Total = 134.5

MULTIPLE INHERITANCE:



The syntax of a derived class with multiple base classes is as follows:

```
class D : visibility B-1, visibility B-2…
{
…
…(body of D)
…
};
```

Where visibility may be either public or private. The base classes are separated by commas.

PROGRAM:

```
#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);
};
Void M :: get_m(int x)
{
m=x;
}
Void N :: get_n(int x)
{
n=y;
}
Void p :: display(void)
```

```
{
Cout<< “m= ”<< m << “\n”;
Cout<< “n= ”<< n << “\n”;
Cout<< “m*n= ”<< m*n << “\n”;
}
int main( )
{
P p;
p.get_m(10);
p.get_n(20);
p.display( );
return 0
}
```

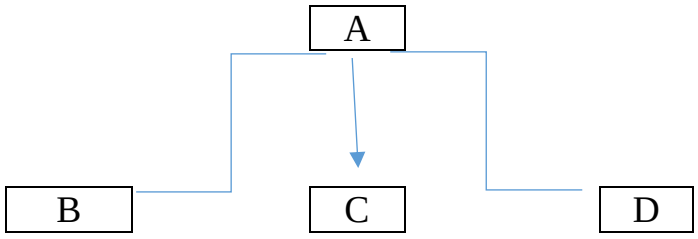
OUTPUT:

m=10

n=20

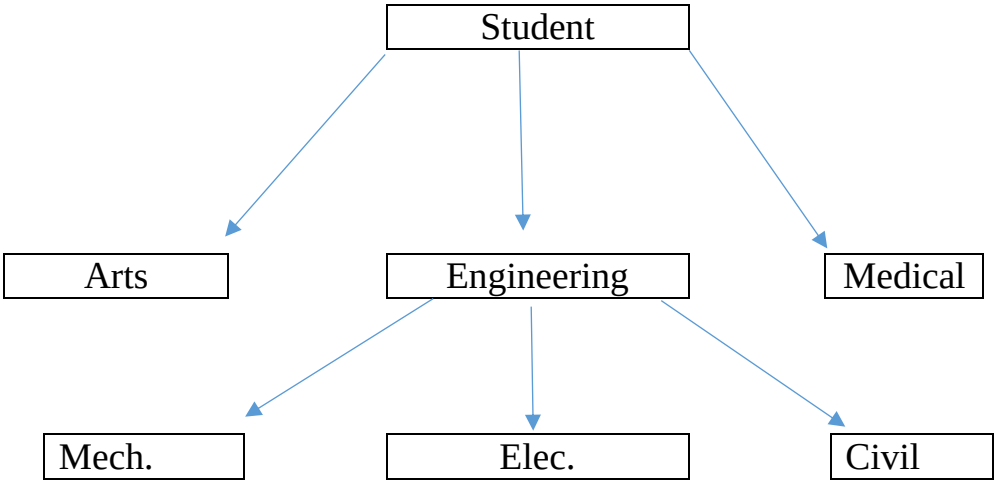
m*n=200

HIERARCHICAL INHERITANCE:



Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming can be cast into a hierarchy where certain features of one level are shared by many others below that level.

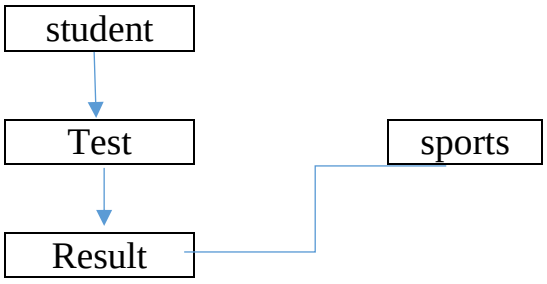
Example: A hierarchical classification of students in a university.



In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.

HYBRID INHERITANCE:

Two or more types of inheritance to design a program. For instance, consider the case of processing the student results discussed. Assume that we have to give weightage for sports before finalizing the results. The weightage for sports is stored in a separate class called sports. The new inheritance relationship between the various classes would be shown



The sports class might look like;

```
class sports
{
protected:
float score;
public:
 void get_score(float);
```

void put_score(void);

};

The result will have both the multilevel and multiple inheritances and its declaration would be as follows:

class result: public test, public sports

{

…

…

};

Where test itself is a derived class from student. That is

class test : public student

{

…

…

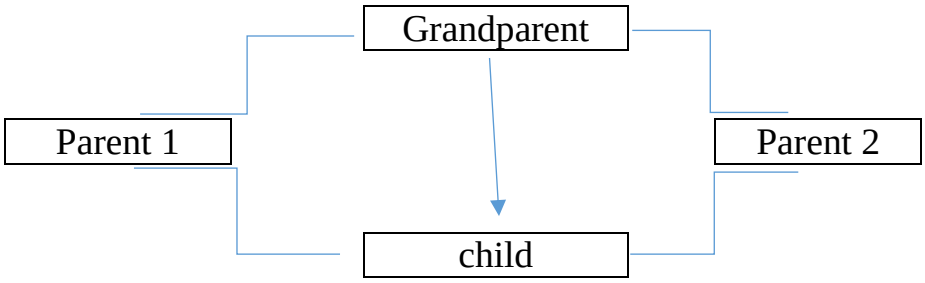
};

LIMITATIONS:

1. In inheritance, the base and inherited classes get tightly coupled. So their independent use is difficult.
2. Wastage of memory and compiler overheads are the drawbacks of inheritance.

VIRITUAL BASE CLASSES:

Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved.



The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual class while declaring the direct or intermediate base classes.

```
Class A
{
…
…
};
Class B1 : virtual public A
{
…
…
};
Class B2 : virtual public A
{
…
…
};
Class C : public B1, public B2
{
…
…
};
```

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

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. It is a design concept in program development and provides a base upon which other classes may be built.

In real programming scenarios, the concept of abstract base classes holds great significance. They are deliberately used in a program for creating derived classes and are not meant for creating instance objects as demos.

The general form of using abstract class:

```
Class vechicle
{
Private:
data-type d1;
data-type d2;
public:
virtual void spec( )=0;
};
Class LMV: public vehicle
{
Public:
Void spec ( )
{
//LMV definition of spec function
}
};
Class HMV : public vehicle
{
Public:
Void spec( )
{
//HMV definition of spec function
}
};
```

```
Class TW: public vehicle
{
Public:
Void spec( )
{
//TW definition of spec function
}
};
```

CONSTRUCTORS IN DERIVED CLASSES:

The constructors play an important role in initializing objects. No base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly in a multi level inheritance, the constructors will be executed in the order of inheritance. The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

The general form of defining a derived constructor is

```
Derived_constructor
base1(arglist1),
base2(arglist2),
…
…
baseN(arglistN),
{
Body of derived constructor
```

}

C++ supports another method of initializing the class objects. This method uses what is known as initialization list in the constructor function.

Constructor (arglist): initialization-section

{

Assignment-section

}

The assignment section is nothing but the body of the constructor function and is used to assign initial values to its data members. The part immediately following the colon is known as the initialization section.

MEMBER CLASSES: NESTING OF CLASSES:

Inheritance is the mechanism of deriving certain properties of one class into another. 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.

class alpha{…};

class beta{…};

class gamma

{

alpha a;

beta b;

…

};

All objects of gamma class will contain the objects a and b. This kind of relationship is called containership or nesting. An independent object is created by its constructor when it is declared with arguments.

class gamma

{

…

alpha a;

beta b;

public:

gamma(arglist): a (arglist1), b (arglist2)

{

}

};

Example:

gamma (int x, int y, float z) : a(x), b(x,z)

{

Assignment section (for ordinary other members)

}

The constructors of the member objects are called in the order in which they are declared in the nested class.