Inheritance

Types of Inheritance

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

Defining derived classes

- Derived class inherits some or all the properties of base class
- General format:

```
class derived-class-name: visibility-mode base-class-name
{
    ……..//
    ……//      members of derived class
    ……//
};
```

Case 1: Base class is privately inherited by a derived class

- The '**public** members' of the base class become '**private** members' of 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.

Case 2: Base class is publicly inherited by a derived class

- The '**public** members' of the base class become '**public** members' of derived class and therefore they are accessible to the objects of the derived class
- In both the cases, **the private members are not inherited** and therefore, the private members of a base class will never become the members of its derived class

Single inheritance (public)

```
#include<iostream>
using namespace std;
class B
{
 int a;
 public:
    int b;
    void get_ab();
    int get_a(void);
   void show_a(void);
};
```

```
class D: public B
{
  int c;
  public:
    void mul(void);
    void display(void);
};
```

```
void B :: get_ab(void){
a=5;b=10; }

int B :: get_a(){
return a; }

void B :: show_a(){
cout <<“a=  “<<a<<“\n”; }

void D :: mul(){
c =b* get_a(); }

void D:: display()
{
cout<<“a =  “<<get_a()<<“\n”;
cout<<“b =  “<<b<<“\n”;
cout<<“c =  “<<c<<“\n”;
}
```

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

Making a private member inheritable

- Private member of a base class cannot be inherited.
- But it is possible to do so with the help of '**protected**' access specifier.
- 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 any other functions outside these two classes.
- When **protected** member is inherited in **public** mode, it becomes **protected** in the **derived** class and is accessible by the member functions of the derived class, also further inheritance is possible.
- When **protected** member is inherited in **private** mode, it becomes **private** in the **derived** class and is accessible by the member functions of the derived class, but further inheritance is impossible.
- In **protected** derivation, **public** and **protected** members become **protected**

```
class alpha
{
   private:                    //optional
   …….                         //visible to member functions within its class
   ……..
   protected:
   …….                         // visible to member functions of its on and derived class
   ……..
   public:
   ……                          //visible to all functions in the program
   ……..
};
```

Access Rights of Derived Classes (or Visibility of inherited members)

Derived class visibility

Base class visibility	private	protected	public
private	-	-	-
protected	private	protected	protected
public	private	protected	public

Example program

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

class Base {
private:
    int pvt = 1;

protected:
    int prot = 2;

public:
    int pub = 3;
    int getPVT()                    // function to access private member
                {
                    return pvt;
                 }
};
```

```
class ProtectedDerived : protected Base {
public:
    int getProt()                       // function to access protected member from Base
        {
          return prot;
        }

    int getPub()                        // function to access public member from Base
        {
           return pub;
        }
 };

int main()
{
  ProtectedDerived object1;
  cout << "Private cannot be accessed." << endl;
  cout << "Protected = " << object1.getProt() << endl;
  cout << "Public = " << object1.getPub() << endl;
  return 0;
}
```

OUTPUT

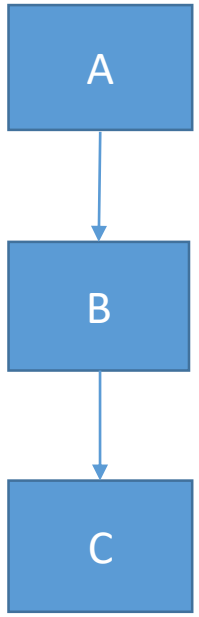
Private cannot be accessed.

Protected = 2

Public = 3

Multilevel Inheritance

- A class can be derived from another derived class.
- For example, class **A** serves as a base class for 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{…..};
class B: public A{…..};
class C: public B{…..};
```

- This process can be extended to any number of levels.

```
#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
{
  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
{
  float total;
  public:
                void display(void);
};
void result::display(void)
{
   total = sub1 + sub2;
  put_number();
  put_marks();
  cout<<“Total = “<<total<<“\n”;
}
```

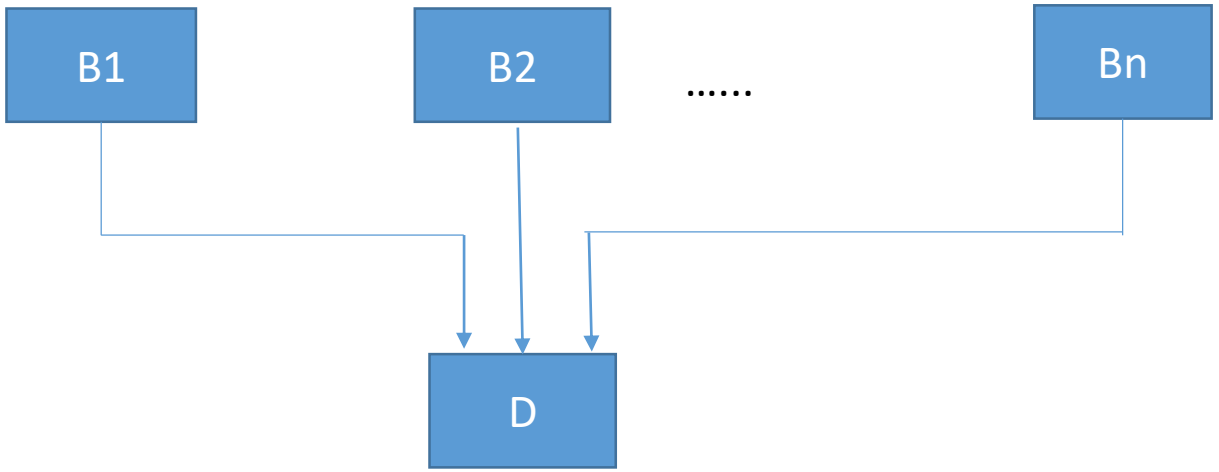
```
 int main()
 {
    result student1;
    student1.get_number(111);
    student1.get_marks(75.0, 59.5);
    student1.display();
    return 0;
}
```

OUTPUT

```
Roll Number: 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5
```

Multiple inheritance

- A class can inherit the attributes of two or more classes. This is known as multiple inheritance.
- Allows us to combine the features of several existing classes as a starting point for defining new classes.
- Let B1, B2,….,Bn be the base classes and D be the derived class.



- Syntax

```
class D: visibility B1, visibility B2 ……
{
        .....
        …….
        ……
};
```

Here visibility specifies whether public or private

Example 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 y)
{
   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
```

Ambiguity resolution in Inheritance

- If a function with the same name appears in more than one base class, then that problem can be solved using a scope resolution operator.

- For example, if a function named display() is present in both base class and derived class, then a simple call to display() will result in invoking the display() in the derived class.(because derived class overrides the base class function).

- In this case, invoke the function using the scope resolution operator in the main function.

```
class A
{ public:
  void display()
  {  cout<<“A\n”;  }
};
class B: public A
{  public:
    void display()
  { cout<<“B\n”;  }
};
int main()
{
 B b;
b.display();
b.A::display();
return 0;
}
```

OUTPUT

B
A

Constructors in Derived Classes

- We can use constructors in derived classes in C++.
- If the base class constructor does not have any arguments, there is no need for any constructor in the derived class.
- But if there are one or more arguments in the base class constructor, derived class need to pass argument to the base class constructor.
- If both base and derived classes have constructors, base class constructor is executed first.

Constructors in Multiple Inheritances

- In multiple inheritances, base classes are constructed in the order in which they appear in the class declaration. For example if there are three classes “A”, “B”, and “C”, and the class “C” is inheriting classes “A” and “B”. If the class “A” is written before class “B” then the constructor of class “A” will be executed first. But if the class “B” is written before class “A” then the constructor of class “B” will be executed first.

Constructors in Multilevel Inheritances

- In multilevel inheritance, the constructors are executed in the order of inheritance. For example if there are three classes “A”, “B”, and “C”, and the class “B” is inheriting classes “A” and the class “C” is inheriting classes “B”. Then the constructor will run according to the order of inheritance such as the constructor of class “A” will be called first then the constructor of class “B” will be called and at the end constructor of class “C” will be called.

Special Syntax

- C++ supports a special syntax for passing arguments to multiple base classes
- The constructor of the derived class receives all the arguments at once and then will pass the call to the respective base classes
- The body is called after the constructors finish executing.

```
Derived_Constructor(arg1,arg2,….):Base1_Constructor(args),Base2_Constructor(args)
{
   Body of derived Constructor
}
```

Example Program

```
#include<iostream>
using namespace std;
class alpha
{
 int x;
 public:
  alpha(int i)
  {
    x = i;
    cout<<“Alpha initialized\n”;
  }
  void show_x(void)
  {
   cout<<“x = “<<x<<“\n”;
  }
};
```

```
class beta
{
  float y;
  public:
     beta(float j)
     {
          y = j;
          cout<<“ Beta initialzed\n”;
     }
      void show_y(void)
     {
       cout<<“y = “<<y<<“\n”;
     }
};
```

```
class gamma : public beta, public alpha
{
  int m,n;
  public:
     gamma(int a, float b, int c, int d):
           alpha(a), beta(b)
    {
       m = c;
       n = d;
      cout<<“gamma initialized\n”;
     }
    void show_mn(void)
    {
      cout<<“m = “ <<m<<“\n”;
      cout<<“n = “ <<n<<“\n”;
    }
};
```

```
int main()
{
  gamma g(5,10.75,20,30);
  cout<<"\n";
  g.show_x();
  g.show_y();
  g.show_mn();
  return 0 ;
}
```

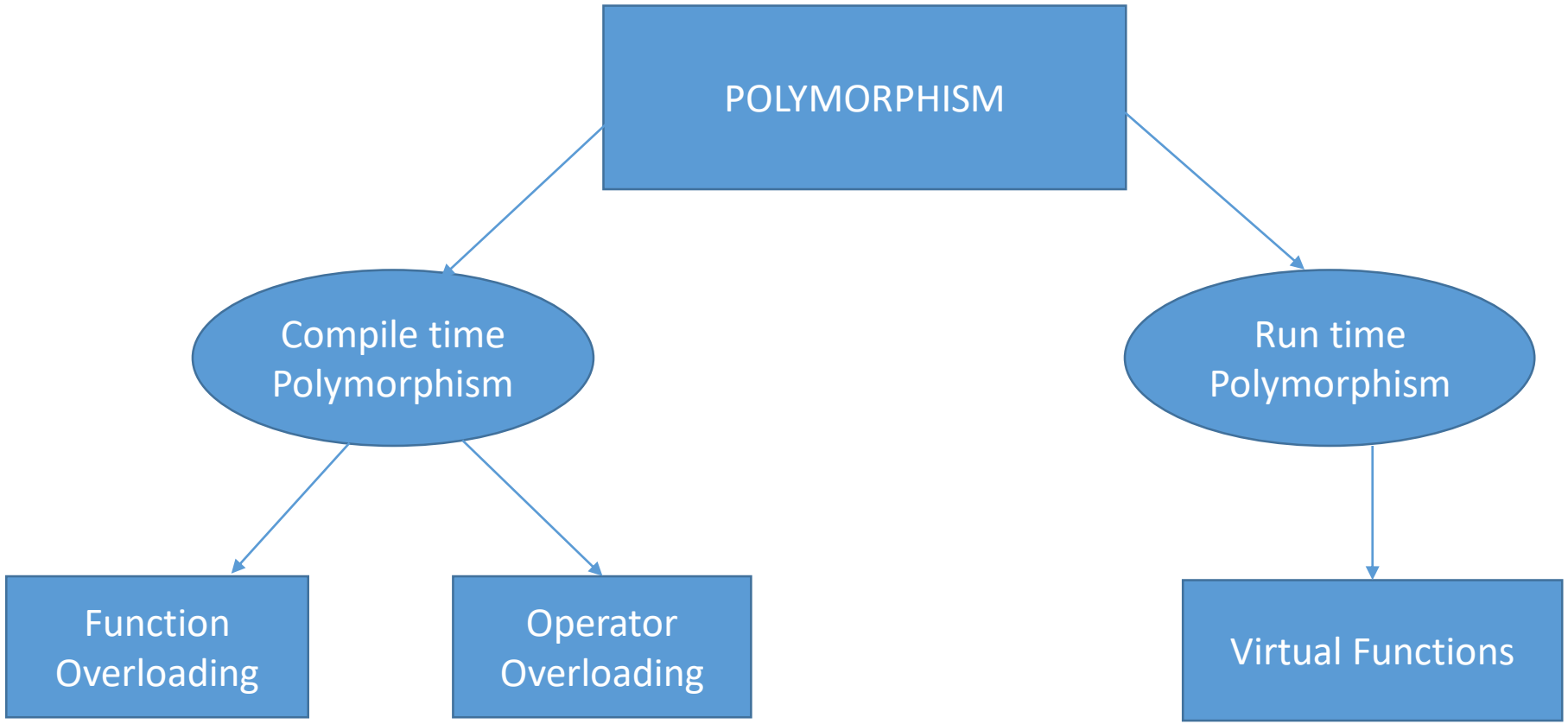
OUTPUT

```
beta initialized
alpha initialized
gamma initialized
x = 5
y = 10.75
m = 20
n = 30
```

ABSTRACT CLASSES

- An abstract class is a class that is not used to create objects.
- It 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.

Polymorphism



Compile time Polymorphism

- The concept of polymorphism is implemented using the overloaded functions and operators.
- The overloaded member functions are selected for invoking by matching arguments, both type and member.
- This information is known to the compiler during compile time and the compiler is able to select the appropriate function for a particular call at the compile time itself.
- This is called early binding/static binding/static linking and this procedure is referred to as **Compile time Polymorphism**. (an object is bound to its function call at its compile time)

Run time Polymorphism

- If the appropriate member function could be selected while the program is running, then it is known as run time polymorphism.
- In C++, run time Polymorphism is achieved by virtual functions.
- Since the function is linked with a particular class much later after the compilation, this process is termed as **late binding**.
- It is also known as **dynamic binding** because the selection of the appropriate function is done dynamically at run time.

Virtual Functions

- When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword **virtual** preceding its normal declaration.

- When a function is made **virtual**, C++ determines which function to use at run time based on the type of object pointed to by the base pointer rather than the type of the pointer.

- By making the base pointer to point to different objects, we can execute different versions of the virtual function.

Example Program

```
#include<iostream>
using namespace std;
class Base
{
 public:
        void display()
        {
        cout<<“\n Display base”;
        }
        virtual void show()
        {
        cout<<“\n Show base”;
        }
};
```

```
class Derived : public Base
{
        public:
                void display()
                    {
                            cout<<"\n Display derived";
                     }
                    void show()
                    {
                            cout<<"\n Show derived";
                    }
};
```

```
int main()
{
 Base B;
 Derived D;
 Base *bptr;
 cout<<"\n bptr points to base\n";
 bptr = &B;
 bptr -> display();
 bptr -> show();
 cout<<"\n\n bptr points to Derived\n";
 bptr = &D;
 bptr -> display();
 bptr -> show();
 return 0;
}
```

OUTPUT

bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived

- When **bptr** is made to point to the object **D**, the statement

  bptr-> display();

  calls only the function associated with the **Base** (that is the **display()** of Base), whereas the statement

  bptr-> show();

  calls the **Derived** version of **show().**

  This is because the function **display()** has not been made **virtual** in the **Base** class.

- We must access the virtual functions through the use of a base class pointer only.

Rules for Virtual Functions

- The virtual functions must be members of some class.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- Virtual function in a base class must be defined, even though it may not be used.
- The prototypes of the base class of a virtual function and all the derived class must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
- We cannot have virtual constructors but we can have virtual destructors.

- While a base pointer can point to any type of the derived object, the reverse is not true. That means, we cannot use a pointer to a derived class to access an object of the base type.
- When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will revoke the base function.

Pure virtual function

- A virtual function, equated to zero is called a pure virtual function.

  virtual void display() = 0;

- It is a function declared in the base class that has no definition relative to the base class.
- A class containing such pure virtual functions cannot be used to declare any objects of its own. Such classes are called abstract classes.
- The main aim of pure virtual functions is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Virtual destructor in C++

- A virtual destructor is used to free up the memory space allocated by the derived class object while deleting instances of the derived class using a base class pointer object.
- They maintain the hierarchy of calling destructors from derived class to base class as the virtual keyword used in the destructor follows the concept of **late binding**.
- The destructor of the base class uses a **virtual** keyword before its name and makes itself a virtual destructor to ensure that the destructor of both the base class and derived class should be called at the run time.
- The derived class's destructor is called first, and then the base class or base class releases the memory occupied by both destructors.

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

class Base {
    public: Base()
            {
                cout << "Constructor of Base class is called" << endl;
            }
   virtual ~Base()
            {
               cout << "Destructor of Base class is called" << endl;
            }
};
```

```
class Child: public Base {
    public: Child()
            {
            cout << "Constructor of Child class is called" << endl;
            }

        ~Child()
             {
              cout << "Destructor of Child class is called" << endl;
            }
};
```

```
int main()
 {
    Base * b = new Child;
    delete b;
    return 0;
 }
```

OUTPUT

```
Constructor of Base class is called
Constructor of Child class is called
Destructor of Child class is called
Destructor of Base class is called
```