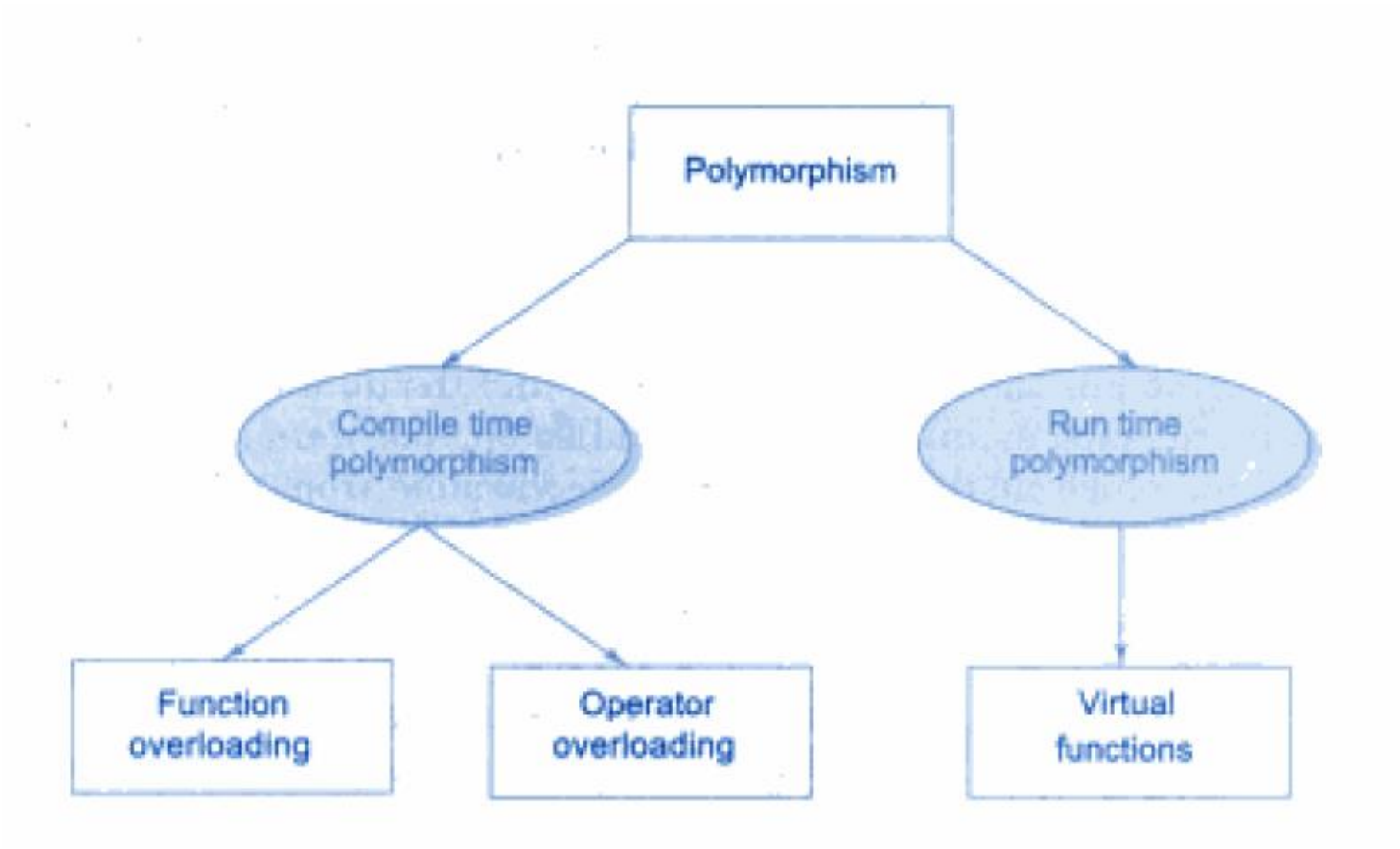
POLYMORPHISM

BY: RICHA JAIN

Polymorphism

- **Polymorphism** is the ability of an object or reference to take many different forms at different instances.
- These are of two types :
  - compile time polymorphism
  - run-time polymorphism

Polymorphism



TYPES OF POLYMORPHISM

- **Compile time polymorphism:** In this method object is bound to the function call at the compile time itself.
- **Run time polymorphism:** In this method object is bound to the function call only at the run time.

Compile time Polymorphism	Run time Polymorphism
It is also known as **Static binding, Early binding** and **overloading** as well.	It is also known as **Dynamic binding, Late binding** and **overriding** as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	**Overriding** is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by **function** overloading and **operator**overloading.	It is achieved by **virtual functions** and **pointers**.
It provides **fast execution** because known early at compile time.	It provides **slow execution** as compare to early binding because it is known at runtime.
Compile time polymorphism is **less flexible** as all things execute at compile time.	Run time polymorphism is **more flexible** as all things execute at run time.

EXAMPLE: Complile time

```
class A
{
   public:
    void func(int x)
 {
 cout << "value of x is " << x <<endl;
   }
        void func(double x)
{
cout << "value of x is " << x <<endl;
   }
     void func(int x, int y)
   {
       cout << "value of x and y is " <<
     << ", " << y << endl;
   }
};
```

```
int main() {

  A obj1;
     obj1.func(7);
    obj1.func(9.132);
    obj1.func(85,64);
    return 0;
}
```

EXAMPLE

```
#include <iostream>
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);
shape = &rec;       // store the address of Rectangle
shape->area();     // call rectangle area.
shape = &tri;      // store the address of Triangle
shape->area();  // call triangle area.
return 0;
}
```

- When the above code is compiled and executed, it produces the following result:

```
Parent class area
Parent class area
```

- 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.

- Now if we precede the declaration of area() in the Shape class with the keyword **virtual** then:

```
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;
 } };
```

- Now the output of the program will be:

```
Rectangle class area
Triangle class area
```

- This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

Virtual function

- When the same function name is used in the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration.
- Virtual means existing in appearance but not in reality.
- Virtual functions are used in late binding or dynamic binding.
- The concept of pointers play important role in the virtual functions.
- When a function is 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 pointer.

- The selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

BINDING

Binding refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses.

Early binding

- Most of the function calls the compiler encounters will be direct function calls. A direct function call is a statement that directly calls a function. For example:

```
 #include <iostream>
void PrintValue(int nValue)
{
   cout << nValue;
}
int main()
{
    PrintValue(5); // This is a direct function call
    return 0;
}
```

Pure Virtual Function

- A pure virtual function is a virtual function which has no body i.e no arguments, no variables and no expressions or statements inside it.

- It's possible to declared a pure virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but has no definition relative to the base class.

Why we use Pure virtual function??

- To make the code simpler.
- To access the members of derived class through the pointer of base class, we need virtual functions.
- But if the virtual function is of no use, because it will not be called then we make it as pure virtual function.

Pure Virtual Function

- Is a function without a body
- Is created by adding the notation '=0' to the virtual function declaration
- Example:

```
virtual int calc_net_salary()=0;
```

Example

```
class shape
{
protected:
int a,b;
public:
void read()
{
cin>>a>>b;
}
virtual void cal_area()=0;
};
class rectangle:public shape
{
void cal_area()
{
double area=a*b;
cout<<area;
}
};
class triangle:public shape
{
void cal_area()
{
double area=(a*b)/2;
cout<<area;
}
};
int main()
{
shape *ptr[2];
rectangle r1;
cout<<“enter leng n bredth”;
r1.read();
```

Example

```
triangle t1;
cout<<“enter base n perpendicular”;
t1.read();
ptr[0]=&r1;
ptr[1]=&t1;
for(int i=0;i<2;i++)
ptr[i]->cal_area();
}
```

OUTPUT

```
Enter length and breadth 10 20
Enter base and perpendicular 5 20
Area of rectangle=200
Area of triangle=50
```

Abstract base class

- The class in which at least one pure virtual function declared is called an abstract base class.
- We can not create object of the abstract class, but we can create pointer to that class.
- A class that does not have a pure virtual function is called a concrete class.

Virtual destructor

- "A destructor is a member function of a class, which gets called when the object goes out of scope". This means all clean ups and final steps of class destruction are to be done in destructor.

- The order of execution of destructor in an inherited class during a clean up is like this.
  1. Derived class destructor
  2. Base class destructor

- In inheritance, when the object of the base class is deleted, the destructor for derived class does not get called at all.

```
#include <iostream.h>
      class Base
      {
             public:
                Base(){ cout<<"Constructor: Base"<<endl;}
                ~Base(){ cout<<"Destructor : Base"<<endl;}
      };
      class Derived: public Base
      {
           //Doing a lot of jobs by extending the functionality
             public:
                 Derived(){ cout<<"Constructor: Derived"<<endl;}
                 ~Derived(){ cout<<"Destructor : Derived"<<endl;}
      };
      void main()
      {
              Base *Var = new Derived();
              delete Var;
      }
```

OUTPUT: Constructing Base

Constructing Derive

Destroying Base

Virtual destructor

- This problem can be fixed by making the base class destructor virtual and this will ensure that the destructor for any class that derives from base will be called.

Virtual destructor

```
class Base
 {
public:
 Base()
{
cout<<"Constructing Base";
}
// this is a virtual destructor:
virtual ~Base()
{
cout<<"Destroying Base";
} };
```

OUTPUT: Constructing Base

```
 Constructing Derive
Destroying Derive
Destroying Base
```