Lecture 13
OOP

Khola Naseem
khola.naseem@uet.edu.pk

Polymorphism:

- Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.
- Inheritance lets us to use attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.
- Polymorphism in C++ means, the same entity (function or object) behaves differently in different scenarios
- For example, think of a base class called Animal that has a method called animalSound().
- Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the Dogs bark, and the cat meows, etc.)

Polymorphism:

- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

:

```
#include <iostream>
using namespace std;
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n";
    }
};
class cat : public Animal {
  public:
    void animalSound() {
      cout << "The cat says: meow meow \n";
    }
};
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};
int main()
{
    Animal a;
    cat c;
    Dog d;
    a.animalSound();
    c.animalSound();
    d.animalSound();
}
```

Output:

```
The animal makes a sound
The cat says: meow meow
The dog says: bow wow
```

Types of Polymorphism in C++:

- **Compile Time Polymorphism**
  - In compile-time polymorphism, a function is called at the time of program compilation. We call this type of polymorphism as early binding or Static binding
  - Function overloading and operator overloading is the type of Compile time polymorphism.
- **Runtime Polymorphism**
  - In a Runtime polymorphism, functions are called at the time the program execution. Hence, it is known as late binding or dynamic binding.
  - Function overriding using a virtual function is a part of runtime polymorphism. In function overriding, more than one method has the same name with same types of the parameter list.
  - It is achieved by using virtual functions and pointers. It provides slow execution as it is known at the run time. Thus, It is more flexible as all the things executed at the run time

Access Overridden Function in C++

- Third method

```
class Base {
   public:
    void print() {
        cout << "Print function in Base class" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Print function in derived class" << endl;
        //Base::print();
    }
};

int main() {
    Base *b;
    Derived d1;
    b=&d1;
    b->print();
   // d1.Base::print();   //access overridden function

    return 0;
```

```
Print function in Base class
```

Virtual Function:

- If it is necessary to use a single pointer to refer to all the different classes' objects. This is because we will have to create a pointer to the base class that refers to all the derived objects.
- But, when the base class pointer contains the derived class address, the object always executes the base class function. For resolving this problem, we use the virtual function.
- When we declare a virtual function, the compiler determines which function to invoke at runtime.
- A virtual function is a member function in the base class. We can redefine it in a derived class. It is part of run time polymorphism. The declaration of the virtual function must be in the base class by using the keyword virtual. A virtual function is not static.
- The virtual function helps to tell the compiler to perform dynamic binding or late binding on the function.

Virtual Function:

- in a Runtime polymorphism, functions are called at the time the program execution. Hence, it is known as late binding or dynamic binding.
- Function overriding is a part of runtime polymorphism. In function overriding, more than one method has the same name with same type of the parameter list.
- A virtual function is declared by keyword virtual. The return type of virtual function may be int, float, void.
- Define a virtual function:

```
class Base {
   public:
    virtual void functionname() {
        // code
    }
};
```

Virtual Function:

- Example:

```
#include <iostream>

using namespace std;

class Base {
   public:
    virtual void print() {
        cout << "Print function in Base class" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Print function in derived class" << endl;
        //Base::print();
    }
};

int main() {
    Base *b;
    Derived d1;
    b=&d1;
    b->print();
   // d1.Base::print();   //access overridden function

    return 0;
}
```

```
Print function in derived class
```

Virtual Function:

➢ Example:

```
class Add
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
     }

    void display ()
    {
    cout<< "display function of base class" <<endl;
    }
};
class Mul: public Add
{
public:
    void print ()
    {
    int x=5;
    int y=10;
    cout<< " Multiplication in drived class:"<<x*y <<endl;
    }

    void display ()
            {
            cout<< "display function of  derived class" <<endl;
            }
};
int main()
{
    Add *ptr;
    Mul m;
    ptr = &m;
    //binded at runtime (Runtime polymorphism) beacuse of virtual function
    ptr->print();
    //binded at compile time a  Non-virtual function
    ptr->display();
    return 0;
}
```

```
 Multiplication in drived class:50
display function of base class
```

Credit: Khola Naseem

Virtual Function:

- C++ override Identifier
- C++ 11 has given us a new identifier override that is very useful to avoid bugs while using virtual functions.

```
class Parent {
   public:
    virtual void display() {
        // code
    }
};

class Derived : public Parent {
   public:
    void display() override {
        // code
    }
};
```

Virtual Function:

- If want to define function outside the class

```
class Derived : public Parent {
   public:
    // function prototype
    void display() override;
};
// function definition
void Derived::display() {
    // code
}
```

Virtual Function:

C++ 11 has given us a new identifier override that is very useful to avoid bugs while using virtual functions.

```
class Add
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
      }

    void display ()
    {
    cout<< "display function of base class" <<endl;
    }
};
class Mul: public Add
{
public:
    void print () override
    {
    int x=5;
    int y=10;
    cout<< " Multiplication in drived class:"<<x*y <<endl;
    }

    void display ()
             {
             cout<< "display function of  derived class" <<endl;
             }
};
int main()
{
    Add *ptr;
    Mul m;
    ptr = &m;
    //binded at runtime (Runtime polymorphism) beacuse of virtual function
    ptr->print();
    //binded at compile time a  Non-virtual function
    ptr->display();
    return 0;
}
```

```
 Multiplication in drived class:50
display function of base class
```

Virtual Function:

- When using virtual functions, it is possible to make mistakes while declaring the member functions of the derived classes.
- Using the override identifier prompts the compiler to display error messages when these mistakes are made.
- Otherwise, the program will simply compile but the virtual function will not be overridden.
- Some of these possible mistakes are:
  - **Functions with incorrect names:**
    - Display() in derived class disply()
  - **Functions with different return types:**
    - Virtual function in parent class return type void in derived class int
  - **Functions with different parameters:**
    - If the parameters of the virtual function and the functions in the derived classes don't match.

Virtual Function:

- Example:

```
#include <iostream>
using namespace std;
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n";
    }
};
class cat : public Animal {
  public:
    void animalSound() {
      cout << "The cat says: meow meow \n";
    }
};
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};
int main()
{
    cat c;
    Dog d;

    Animal* ac=&c;
    Animal* ad=&d;

    ac->animalSound();
    ad->animalSound();

}
```

```
The animal makes a sound
The animal makes a sound
```

Credit: Khola Naseem

Virtual Function:

- Example:

```cpp
#include <iostream>
using namespace std;
// Base class
class Animal {
  public:
    virtual void animalSound() {
      cout << "The animal makes a sound \n";
    }
};
class cat : public Animal {
  public:
    void animalSound() override{
      cout << "The cat says: meow meow \n";
    }
};
class Dog : public Animal {
  public:
    void animalSound() override {
      cout << "The dog says: bow wow \n";
    }
};


int main()
{
    cat c;
    Dog d;
    Animal* ac=&c;
    Animal* ad=&d;
    ac->animalSound();
    ad->animalSound();

}
```

```
The cat says: meow meow
The dog says: bow wow
```

Polymorphism:

- Example:

```
class Shape {
   protected:
      int width, height;

   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      int area() {
         cout << "Area of parent class :" << width * height << endl;
         return width * height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   shape = &t;
   shape->area(); //Triangle class
```

```
Area of parent class :6
Area of parent class :32
```

Polymorphism:

- Example:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area() {
         cout << "Area of parent class :" << width * height << endl;
         return width * height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   shape = &t;
   shape->area(); //Triangle class
   return 0;
}
```

```
Area of rectangle class :6
Area of triangle class :16
```

Overriding vs Overloading:

Function Overloading	Function Overriding
Function overloading in C++ can occur with or without inheritance.	Function overriding in C++ can only occur in the presence of inheritance.
Overloaded functions must have different function signatures i.e., the number of parameters or the data type of parameters should be different.	Overridden functions must have the same function signature i.e., the number of parameters as well as their data type must be he same.
It represents the compile-time polymorphism or early binding as overloading occurs during compile time.	It represents the run-time polymorphism or late binding as overriding occurs during run time.
Overloading takes place within the same class	Overriding occurs in a parent class and its child class.
No special keyword is used to overload a function.	Virtual keyword in the base class and Override keyword in the derived class can be used to override a function.
Overloading is done to acquire the different behavior to the same function depending on the arguments passed to the functions.	Overriding is done when the derived class function is expected to perform differently than the base class function.

Polymorphism:

- pointer to a base class is used to store the address of a derived class object; in fact, you can use a pointer to any direct or indirect base class to store the address of a derived class object. Figure shows how the Carton class is derived from the Box base class by single inheritance, and the CerealPack class is derived by multiple inheritances from the Carton and FoodContainer base classes. It illustrates how pointers to base classes can be used to store addresses of derived class objects

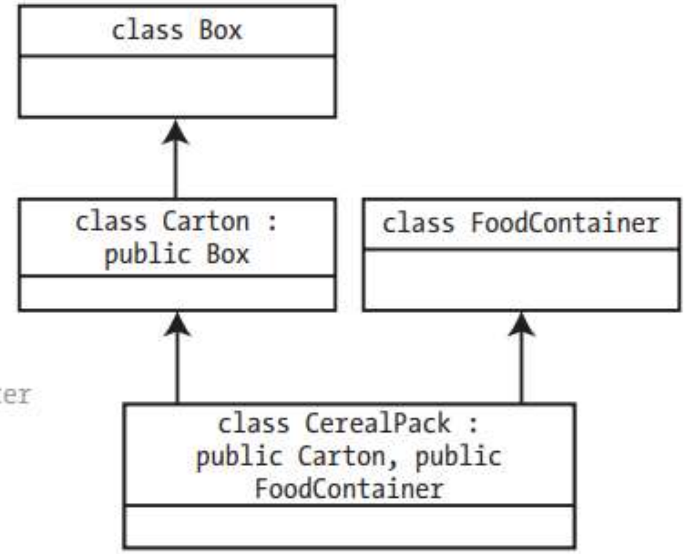
```
CerealPack cereal;

// You can store the address of
// the cereal object
// in any base class pointer:

Carton* pCarton {&cereal};
Box* pBox {&cereal};
FoodContainer* pFood {&cereal};

// You can store the address of
// a Carton object in a base pointer

Carton carton;
pBox = &carton;
```



Polymorphism:

- The reverse is not true.
- For instance, you can’t use a pointer of type Carton* to store the address of an object of type Box.
- This is logical because a pointer type incorporates the type of object to which it can point.
- A derived class object is a specialization of its base—it is a base class object—so using a pointer to the base to store its address is reasonable.
- However, a base class object is definitely not a derived class object, so a pointer to a derived class type cannot point to it

Polymorphism:

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

class Add
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
      }
    void display ()
    {
    cout<< "display function of base class" <<endl;
    }
};
class Mul: public Add
{
public:
    void print () override{
    int x=5;
    int y=10;
    cout<< " Multiplication in drived class:"<<x*y <<endl;
    }

    void display ()
            {
            cout<< "display function of  derived class" <<endl;
            }
};
int main()
{
   Mul *k;
   Add a;
   k=&a;

    k->print();
    //binded at compile time a  Non-virtual function
    k->display();
    return 0;
}
```

```
[Warning] override controls (override/final) only available with -std=c++11 or -std=gnu++11
In function 'int main()':
[Error] invalid conversion from 'Add*' to 'Mul*' [-fpermissive]
```

Credit: Khola Naseem

Polymorphism:

- For a function to behave "virtually," its definition in a derived class must have the same signature as it has in the base class. If the base class function is const, for instance, then the derived class function must therefore also be const.
- If a member function definition is outside the class definition, you must not add the virtual keyword to the function definition; it would be an error to do so. You can only add virtual to declarations or definitions inside a class definition.

```
class Derived : public Parent {
   public:
    // function prototype
    void display() override;
};
// function definition
void Derived::display() {
    // code
}
```

Polymorphism:

- **Virtual Functions and Class Hierarchies**
- If you want your function to be treated as virtual when it is called using a base class pointer, then you must declare it as virtual in the base class. You can have as many virtual functions as you want in a base class

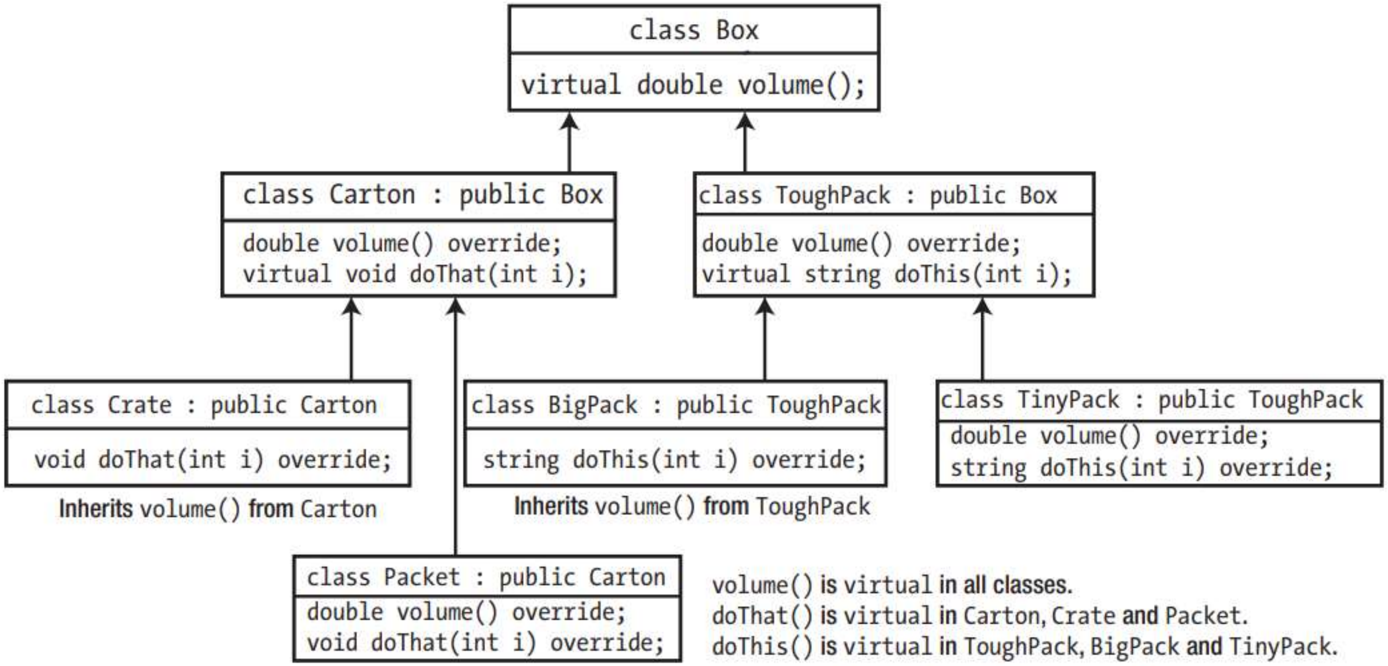


Figure 15-3. Virtual functions in a hierarchy

Polymorphism:

- **Virtual Functions and Class Hierarchies**
- When you specify a function as virtual in a class, the function is virtual in all classes derived directly or indirectly from that class. All of the classes derived from the Box class inherit the virtual nature of the volume() function, even if they do not repeat the virtual keyword.
- You can call volume() for objects of any of these class types through a pointer of type Box* because the pointer can contain the address of an object of any class in the hierarchy.
- The Crate class doesn't define volume(), so the version inherited from Carton would be called for Crate objects. It is inherited as a virtual function and therefore can be called polymorphically.

Polymorphism:

- **Virtual Functions and Class Hierarchies**
- When you specify a function as virtual in a class, the function is virtual in all classes derived directly or indirectly from that class. All of the classes derived from the Box class inherit the virtual nature of the volume() function, even if they do not repeat the virtual keyword.
- You can call volume() for objects of any of these class types through a pointer of type Box* because the pointer can contain the address of an object of any class in the hierarchy.
- The Crate class doesn't define volume(), so the version inherited from Carton would be called for Crate objects. It is inherited as a virtual function and therefore can be called polymorphically.
- A pointer carton, of type Carton*, could also be used to call volume(), but only for objects of the Carton class and the two classes that have Carton as a base: Crate and Packet.

Polymorphism:

- **Virtual Functions and Class Hierarchies**
- When you specify a function as virtual in a class, the function is virtual in all classes derived directly or indirectly from that class. All of the classes derived from the Box class inherit the virtual nature of the volume() function, even if they do not repeat the virtual keyword.
- The Carton class and the classes derived from it also contain the virtual function doThat(). This function can also be called polymorphically using a pointer of type Carton*.
- Of course, you cannot call doThat() for these classes using a pointer of type Box* because the Box class doesn’t define the function doThat().

Polymorphism:

➢ Even when the print function is not override in the child class

```
class derived1: public parent
{
public:
    void print () override{
    int x=5;
    int y=10;
    cout<< " Multiplication in derived class:"<<x*y <<endl;
    }
    void display ()
    {
     cout<< "display function of  derived1 class" <<endl;
    }
};
class derived2: public derived1
{
public:


    void display ()
    {
        cout<< "display function of  derived2 class" <<endl;
    }
};
int main()
{
   parent *ptr;
   derived1 m;
   ptr = &m;
   ptr->print();
   ptr->display();

   derived2 d;
   ptr = &d;
   ptr->print();
   ptr->display();
    return 0;
}
```

E:\UET\Spring 23\OOP\Class\ploymorphisim.exe

```
 Multiplication in derived class:50
display function of base class
 Multiplication in derived class:50
display function of base class

--------------------------------
Process exited after 0.1278 seconds with return value 0
Press any key to continue . . .
```

Polymorphism:

```
class parent
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
      }
    void display ()
    {
    cout<< "display function of base class" <<endl;
    }
};
class derived1: public parent
{
public:
    void print () override{
    int x=5;
    int y=10;
    cout<< " Multiplication in drived class:"<<x*y <<endl;
    }
    void display ()
    {
     cout<< "display function of  derived1 class" <<endl;
    }
};
class derived2: public derived1
{
public:
    void print () override{
    int x=10;
    int y=5;
    cout<< " divison in derived2 class:"<<x/y <<endl;
    }

    void display ()
          {
          cout<< "display function of  derived2 class" <<endl;
          }
};
```

```
int main()
{
   parent *ptr;
   derived1 m;
   ptr = &m;
   ptr->print();
   ptr->display();
   derived2 d;
   ptr = &d;
   ptr->print();
   ptr->display();
    return 0;
}
```

Output:

```
 Multiplication in derived class:50
display function of base class
 divison in derived2 class:2
display function of base class
```

Polymorphism:

```cpp
class derived1: public parent
{
public:
    void print () override{
    int x=5;
    int y=10;
    cout<< " Multiplication in derived class:"<<x*y <<endl;
    }
   virtual void display ()
    {
     cout<< "display function of  derived1 class" <<endl;
    }
};
class derived2: public derived1
{
public:
    void print () override{
    int x=10;
    int y=5;
    cout<< " divison in derived2 class:"<<x/y <<endl;
    }

    void display () override
    {
        cout<< "display function of  derived2 class" <<endl;
    }
};
int main()
{
   parent *ptr;
   derived1 m;
   ptr = &m;
   ptr->print();
   ptr->display();
   derived2 d;
   ptr = &d;
   ptr->print();
   ptr->display();
    return 0;
}
```

```
 Multiplication in derived class:50
display function of base class
 divison in derived2 class:2
display function of base class

--------------------------------
Process exited after 0.138 seconds with return value 0
Press any key to continue . . .
```

Polymorphism:

```cpp
class derived1: public parent
{
public:
    void print () override{
    int x=5;
    int y=10;
    cout<< " Multiplication in derived class:"<<x*y <<endl;
    }
    virtual void display ()
    {
     cout<< "display function of  derived1 class" <<endl;
    }
};
class derived2: public derived1
{
public:
    void print () override{
    int x=10;
    int y=5;
    cout<< " divison in derived2 class:"<<x/y <<endl;
    }

    void display () override
    {
        cout<< "display function of  derived2 class" <<endl;
    }
};
int main()
{
   parent *ptr;
   derived1 m;
   ptr = &m;
   ptr->print();
   ptr->display();
   derived1 *dd;
   derived2 d;
   dd = &d;
   ptr->print();
   dd->display();
   return 0;
}
```

Select E:\UET\Spring 23\OOP\Class\ploymorphisim.exe

```
 Multiplication in derived class:50
display function of base class
 Multiplication in derived class:50
display function of  derived2 class

--------------------------------
Process exited after 0.1199 seconds with return
Press any key to continue . . .
```

polymorphism:

- Using final
- Sometimes you may want to prevent a member function from being overridden in a derived class.
- This could be because you want to limit how a derived class can modify the behavior of the class interface, for example. You can do this by specifying that a function is final. You could prevent the volume() function in the Carton class from being overridden by definitions in classes derived from Carton by specifying it like this:

```
class Carton : public Box
{
public:
  double volume() const override final
  {
    // Function body as before...
  }

  // Details of the class as in Ex15_02...
};
```

- Attempts to override volume() in classes that have Carton as a base will result in a compiler error. This ensures that only the Carton version can be used for derived class objects.

polymorphism:

- Using final

```
class parent
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
      }
};
class derived1: public parent
{
public:
    void print () override final{
    int x=5;
    int y=10;
    cout<< " Multiplication in derived class:"<<x*y <<endl;
    }
};
class derived2: public derived1
{
public:
    void print () override{
    int x=10;
    int y=5;
    cout<< " divison in derived2 class:"<<x/y <<endl;
    }
};
int main()
{
   parent *ptr;
   derived1 m;
   ptr = &m;
   ptr->print();
   derived2 d;
   ptr = &d;
   ptr->print();

    return 0;
}
```

Error:

```
[Warning] override controls (override/final) only available with -std=c++11 or -std=gnu++11
[Error] virtual function 'virtual void derived2::print()'
[Error] overriding final function 'virtual void derived1::print()'
recipe for target 'ploymorphisim.o' failed
```

Credit: Khola Naseem

polymorphism:

- Using final

```
class parent
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
      }
};
class derived1: public parent
{
public:
    void print () override final{
    int x=5;
    int y=10;
    cout<< " Multiplication in derived class:"<<x*y <<endl;
    }
};
class derived2: public derived1
{
public:
    void print () override{
    int x=10;
    int y=5;
    cout<< " divison in derived2 class:"<<x/y <<endl;
    }
};
int main()
{
   parent *ptr;
   derived1 m;
   ptr = &m;
   ptr->print();
   derived2 d;
   ptr = &d;
   ptr->print();

    return 0;
}
```

Error:

```
[Warning] override controls (override/final) only available with -std=c++11 or -std=gnu++11
[Error] virtual function 'virtual void derived2::print()'
[Error] overriding final function 'virtual void derived1::print()'
recipe for target 'ploymorphisim.o' failed
```

Credit: Khola Naseem

Access Specifiers and Virtual Functions :

- The access specification of a virtual function in a derived class can be different from the specification in the base class.
- When you call the virtual function through a base class pointer, the access specification in the base class determines whether the function is accessible, regardless of the type of object pointed to.
- If the virtual function is public in the base class, it can be called for any derived class through a pointer (or a reference) to the base class, regardless of the access specification in the derived class

Access Specifiers and Virtual Functions :

- The access specification of a virtual function in a derived class

```
class parent
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
     }
};
class derived1: public parent
{
public:
    void print () override {
    int x=5;
    int y=10;
    cout<< " Multiplication in derived class:"<<x*y <<endl;
    }
};
class derived2: public derived1
{
private:
    void print () override{
    int x=10;
    int y=5;
    cout<< " divison in derived2 class:"<<x/y <<endl;
    }
};
int main()
{
   parent *ptr;
   derived1 m;
   ptr = &m;
   ptr->print();
   derived2 d;
   ptr = &d;
   ptr->print();
    return 0;
}
```

```
Multiplication in derived class:50
divison in derived2 class:2

--------------------------------
Process exited after 0.1285 seconds with return value 0
Press any key to continue . . .
```

Default Argument Values in Virtual Functions:

- Default argument values are dealt with at compile time, so you can get unexpected results when you use default argument values with virtual function parameters.
- If the base class declaration of a virtual function has a default argument value and you call the function through a base pointer, you'll always get the default argument value from the base class version of the function.
- Any default argument values in derived class versions of the function will have no effect.

Default Argument Values in Virtual Functions:

- Default argument values

```cpp
class parent
{
public:
    virtual void print (int i=10)
    {
    cout<<"i is in base class"<<i<<endl;
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
     }
};
class derived1: public parent
{
public:
    void print (int i=5) override {
    cout<<"i is in derive1 class"<<i<<endl;
    int x=5;
    int y=10;
    cout<< " Multiplication in derived class:"<<x*y <<endl;
    }
};
class derived2: public derived1
{
private:
    void print (int i=50) override{
    cout<<"i is in derived2  class"<<i<<endl;
    int x=10;
    int y=5;
    cout<< " divison in derived2 class:"<<x/y <<endl;
    }
};
int main()
{
   parent *ptr;
   derived1 m;
   ptr = &m;
   ptr->print();
   deriv
   ptr =
   ptr->print();
    return 0;
```

1/ 3 private void derived2::print (int i=50)

E:\UET\Spring 23\OOP\Class\ploymorphisim.exe

```
i is in derive1 class10
 Multiplication in derived class:50
i is in derived2  class10
 divison in derived2 class:2

--------------------------------
Process exited after 0.0743 seconds with return value 0
Press any key to continue . . .
```

Credit: Khola Naseem

Using References to Call Virtual Functions

- You can call a virtual function through a reference; reference parameters are particularly powerful tools for applying polymorphism, particularly when calling functions that use pass-by-reference.
- You can pass a base class object or any derived class object to a function with a parameter that’s a reference to the base class. You can use the reference parameter within the function body to call a virtual function in the base class and get polymorphic behavior.
- When the function executes, the virtual function for the object that was passed as the argument is selected automatically at runtime

Using References to Call Virtual Functions

```
class parent
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
      }
};
class derived1: public parent
{
public:
    void print () override {
    int x=5;
    int y=10;
    cout<< " Multiplication in derived class:"<<x*y <<endl;
    }
};
class derived2: public derived1
{
public:
    void print () override{
    int x=10;
    int y=5;
    cout<< " divison in derived2 class:"<<x/y <<endl;
    }
};
void callprint(parent &parent)
{
    parent.print();
}
int main()
{
   derived1 d1;
   derived2 d2;
   callprint(d1);
   cout<<"pass d2"<<endl;
   callprint(d2);
    return 0;
}
```

```
 Multiplication in derived class:50
pass d2
 divison in derived2 class:2

--------------------------------
Process exited after 0.1207 seconds with return value 0
Press any key to continue . . .
```

Credit: Khola Naseem

The Cost of Polymorphism:

- As you know, there's no such thing as a free lunch, and this also applies to polymorphism. You pay for polymorphism in two ways: it requires more memory, and virtual function calls result in additional runtime overhead. These consequences arise because of the way that virtual function calls are typically implemented in practice.
- Luckily, both costs are mostly marginal at best and can mostly be ignored. For instance, suppose two classes, A and B, contain identical member variables, but A contains virtual functions, whereas B's functions are all nonvirtual. In this case, an object of type A requires more memory than an object of type B.
- You can create a simple program with two such class objects and use the sizeof operator to see the difference in memory occupied by objects with and without virtual functions.

The Cost of Polymorphism:

- As you know, there's no such thing as a free lunch, and this also applies to polymorphism. You pay for polymorphism in two ways: it requires more memory, and virtual function calls result in additional runtime overhead. These consequences arise because of the way that virtual function calls are typically implemented in practice.
- Luckily, both costs are mostly marginal at best and can mostly be ignored. For instance, suppose two classes, A and B, contain identical member variables, but A contains virtual functions, whereas B's functions are all nonvirtual. In this case, an object of type A requires more memory than an object of type B.
- You can create a simple program with two such class objects and use the sizeof operator to see the difference in memory occupied by objects with and without virtual functions.

The Cost of Polymorphism:

- The reason for the increase in memory is that when you create an object of a polymorphic class type, a special pointer is created in the object.
- This pointer is used to call any of the virtual functions in the object. The special pointer points to a table of function pointers that gets created for the class.

Pure virtual function:

- Pure virtual functions are used
  - if a function doesn't have any use in the base class
  - but the function must be implemented by all its derived classes
- Run time polymorphism
- Example: Suppose, we have derived Triangle and Rectangle classes from the Shape class, and we want to calculate the area of all these shapes.
- In this case, we can create a pure virtual function named area() in the Shape. Since it's a pure virtual function, all derived classes Triangle and Rectangle must include the area() function with implementation.
- A pure virtual function doesn't have the function body and it must end with = 0
- Syntax:
  - class Shape

```
{ public:
 virtual void area() = 0;
  };
```

Pure virtual function:

- Example:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area() =0;
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   shape = &t;
   shape->area(); //Triangle class
   return 0;
}
```

Output:

```
Area of rectangle class :6
Area of triangle class :16
```

Abstract Class:

- A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.
- We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions).
-

```
class Shape {
  protected:
    int width, height;
  public:
    Shape( int c = 0, int d = 0){
      width = c;
      height = d;
    }
    virtual int area()=0;
    int getHeight()
    {
      return height;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int c = 0, int d = 0):Shape(c, d) { }
    int area () override {
      cout << "Area of rectangle class :" << width * height << endl;
      return (width * height);
    }
};
class Triangle: public Shape {
  public:
    Triangle( int c = 0, int d = 0):Shape(c, d) { }

    int area () override {
      cout << "Area of triangle class :" << (width * height)/2 << endl;
      return (width * height / 2);
    }
};
int main() {
  Shape *shape;
  Shape s;
  Rectangle r(3,2);
  Triangle  t(4,8);
  shape = &r;
  shape->area(); //rectangle class
  cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
  shape = &t;
  shape->area(); //Triangle class
  cout<<"Get the height of Triangle: "<<shape->getHeight();
  return 0;
}
```

Error

```
[Error] cannot declare variable 's' to be of abstract type 'Shape'
[Note] because the following virtual functions are pure within 'Shape':
[Note] virtual int Shape::area()
```

Abstract Class:

- A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.
- We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions) .
- ```
class Shape {
  protected:
    int width, height;
  public:
    Shape(int c = 0, int d = 0){
      width = c;
      height = d;
    }
    virtual int area() =0;
    int getHeight()
    {
      return height;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle(int c = 0, int d = 0):Shape(c, d) { }
    int area () override {
      cout << "Area of rectangle class :" << width * height << endl;
      return (width * height);
    }
};
class Triangle: public Shape {
  public:
    Triangle(int c = 0, int d = 0):Shape(c, d) { }

    int area () override {
      cout << "Area of triangle class :" << (width * height)/2 << endl;
      return (width * height / 2);
    }
};
int main() {
  Shape *shape;
  Rectangle r(3,2);
  Triangle t(4,8);
  shape = &r;
  shape->area(); //rectangle class
  cout<<"Get height for rectangle: "<<shape->getHeight()<<endl;
  shape = &t;
  shape->area(); //Triangle class
  cout<<"Get height for Triangle"<<shape->getHeight();
  return 0;
}
```

Output:

```
Area of rectangle class :6
Get the height of rectangle: 2
Area of triangle class :16
Get the height of Triangle: 8
```

Credit: Khola Naseem
```

Abstract Class:

- In this case, we can create a pure virtual function named area() in the Shape. Since it's a pure virtual function, all derived classes Triangle and Rectangle must include the area() function with implementation. otherwise, the derived class is also abstract.
-

```
class Shape {
  protected:
    int width, height;
  public:
    Shape( int c = 0, int d = 0){
      width = c;
      height = d;
    }
    virtual int area() =0;
    int getHeight()
    {
     return height;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int c = 0, int d = 0):Shape(c, d) { }
    /*int area () override {
      cout << "Area of rectangle class :" << width * height << endl;
      return (width * height);
    }*/
};
class Triangle: public Shape {
  public:
    Triangle( int c = 0, int d = 0):Shape(c, d) { }

    int area () override {
      cout << "Area of triangle class :" << (width * height)/2 << endl;
      return (width * height / 2);
    }
};
int main() {
  Shape *shape;
  Rectangle r(3,2);
  Triangle  t(4,8);
  shape = &r;
  shape->area(); //rectangle class
  cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
  shape = &t;
  shape->area(); //Triangle class
  cout<<"Get the height of Triangle: "<<shape->getHeight();
  return 0;
}
```

```
In function 'int main()':
[Error] cannot declare variable 'r' to be of abstract type 'Rectangle'
[Note] because the following virtual functions are pure within 'Rectangle':
[Note] virtual int Shape::area()
```

Abstract Class:

- **Example:**
- Creating Abstract Base Class Employee
- Class Employee provides functions earnings and print, in addition to various get and set functions that manipulate Employee's data members. An earnings function certainly applies generally to all employees, but each earnings calculation depends on the employee's class. So we declare earnings as pure virtual in base class Employee because a default implementation does not make sense for that function—there is not enough information to determine what amount earnings should return. Each derived class overrides earnings with an appropriate implementation. To calculate an employee's earnings.

Virtual VS pure virtual:

- Example:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area()
      {
        cout<<"shape area \n";
      }
      int getHeight()
      {
        return height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }
      /*int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }*/
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
   shape = &t;
   shape->area(); //Triangle class
   cout<<"Get the height of Triangle: "<<shape->getHeight();
   return 0;
}
```

Output:

```
shape area
Get the height of rectangle: 2
Area of triangle class :16
Get the height of Triangle: 8
```

Credit: Khola Naseem

Virtual VS pure virtual:

- Example:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area()
      {
         cout<<"shape area \n";
      }
      int getHeight()
      {
         return height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }
      int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Shape s;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
   shape = &t;
   shape->area(); //Triangle class
   cout<<"Get the height of Triangle: "<<shape->getHeight();
   return 0;
}
```

Output:

```
Area of rectangle class :6
Get the height of rectangle: 2
Area of triangle class :16
Get the height of Triangle: 8
```

Reference material

For Practice Questions, refer to these books

- C++ Programming From Problem Analysis To Program Design, 5th Edition, D.S.Malik. Chapter 12.
- C++ How to Program, Deitel & Deitel, 5th Edition, Prentice Hall.
- Object Oriented Programming in C++ by Robert Lafore.
- Object Oriented Software Construction, Bertrand Meyer's
- Object-Oriented Analysis and Design with applications, Grady Booch et al, 3Rd Edition, Pearson, 2007
- Web