COEN 244

POLYMORPHISM

Compile-time binding (Static binding)

- Compile-time binding is to associate a function’s name with the entry point of the function at compile time.
- In C, only compile-time binding is provided.
- In C++, all non-virtual functions are bound at compile-time.

Static binding - Example

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

void sayHi();

int main(){
	sayHi();	// the compiler binds any invocation of sayHi()
			// to sayHi()’s entry point.
}

void sayHi(){
	cout << “Hello, World!\n”;
}
```

Run-time binding (Dynamic binding)

- Run-time binding is to associate a function's name with the entry point at run-time.
- C++ supports run-time binding through virtual functions.
- Polymorphism is thus implemented by virtual functions and run-time binding mechanism in C++. A class is called **polymorphic** if it contains virtual functions.

Dynamic binding - Example

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

class Shape{
public:
   virtual void sayHi() { cout <<‘‘Just hi! \n’’;}
};
class Triangle : public Shape{
public:
   virtual void sayHi() { cout <<‘‘Hi from a triangle! \n’’;}
};
class Rectangle : public Shape{
public:
   virtual void sayHi() { cout <<‘‘Hi from a rectangle! \n; }
};
```

```
int main(){
   Shape *p;
   int which;
   cout << ‘‘1 -- shape, 2 -- triangle, 3 -- rectangle\n ’’;
   cin >> which;
   switch ( which ) {
   case 1:  p = new Shape; break;
   case 2:  p = new Triangle; break;
   case 3:  p = new Rectangle; break;
   }
   p -> sayHi();    // dynamic binding of sayHi()
   delete p;
}
```

Virtual Functions

- To declare a function virtual, we use the Keyword virtual.

```
class Shape{
public:
  virtual void sayHi (){ cout <<‘‘Just hi! \n’’;}
};
```

- If the member function definition is outside the class, the keyword virtual must not be specified again.

```
class Shape{
public:
  virtual void sayHi ();
};
virtual void Shape::sayHi (){  // error
  cout << ‘‘Just hi! \n’’;
}
```

Virtual Functions

- A virtual function can be used same as non-virutal member functions.

```
class B {
public:
   virtual void m() { cout << ‘‘Hello! \n’’; }
};
int main(){
   B b_obj;
   b_obj.m();
}
```

- A virtual function can be inherited from a base class by a derived class, like other class member functions.

```
class B {
public:
   virtual void m() { cout << ‘‘Hello! \n’’;}
};
class D : public B{
   // inherite B::m()
};
int main(){
   D d_obj;
   d_obj.m();
}
```

Virtual Functions

- To let derived classes have their own implementation for the virtual function. We override base class virtual functions in derived class.

```
class Shape{
public:
   virtual void sayHi() { cout <<‘‘Just hi! \n’’;}
};
class Triangle : public Shape{
public:
   // overrides Shape::sayHi(), automatically virtual
   void sayHi() { cout <<‘‘Hi from a triangle! \n’’;}
};
```

Polymorphism

- Dynamic binding is enabled when a virtual function is invoked through a derived class object which is refered indirectly by either a base class pointer or reference.

```
void print(Shape obj, Shape *ptr, Shape &ref){
   ptr -> sayHi();    // bound at run time
   ref.sayHi();       // bound at run time
   obj.sayHi();       // bound at compile time
}

int main(){
  Triangle mytri;
  print( mytri, &mytri, mytri );
}
```

Exercise 1:

- Is **m()** bound at compile time or run time?

```
class B {
public:
   void m() { cout << ‘‘B::m \n’’;}
};

class D : public B{
public:
   void m() { cout << ‘‘D::m \n’’;}
};

int main(){
    B *p;
    p = new D;
    p -> m();
}
```

Exercise 2:

- Is **m()** bound at compile time or run time?

```
class B {
public:
   virtual void m() { cout << ‘‘B::m \n’’;}
};

class D : public B{
public:
   void m() { cout << ‘‘D::m \n’’;}
};

int main(){
    D d;
    d.m();
}
```

Static Invocation of Virtual Functions

- We can override the virtual mechanism when using the class scope operator to invoke a virtual function. Thus, the virtual function is resolved at compile-time.

```
class B {
public:
   virtual void m(){ cout << ‘‘B::m \n’’;}
};

class D : public B{
public:
   void m(){ cout << ‘‘D::m \n’’;}
};

int main(){
   B *p = new D;
   p -> B::m();
}
```

Run-time v.s. compile-time binding

- The approach of using inheritance and run-time binding facilitates the following software quality factors:
  - Reuse
  - Transparent extensibility
  - Delaying decisions until run-time

When to choose use different kinds of bindings:

- Use compile-time binding when you are sure that any derived class will not want to override the function dynamically.
- Use run-time binding when the derived class may be able to provide a different implementation that should be selected at run-time.

Example

```
class Shape {
public:
  void setDim(double, double = 0);
  virtual void showArea();
protected:
  double x, y;
};

void Shape::setDim(double xx, double yy) : x(xx), y(yy){}

void Shape::showArea(){
  cout << "No area computation defined for this class\n";
}
```

Example - (Cont'd)

```
// Derived class Triangle from Shape
class Triangle : public Shape{
public:
  virtual void showArea();
};
void Triangle::showArea(){
    cout << "Triangle with height " << x << " and base " << y
         << " has an area of " << x * y* 0.5 << endl;
}

// Derived class Rectangle from Shape
class Rectangle : public Shape{
public:
    virtual void showArea();
};
void Rectangle::showArea(){
    cout << "Rectangle with dimentions " << x << " and " << y
         << " has an area of " << x * y << endl;
}
```

```
// Derived class Circle from Shape
class Circle : public Shape{
public:
   virtual void showArea();
};
void Circle::showArea(){
    cout << "Circle with radius " << x
         << " has an area of " << 3.14 * x * x << endl;
}
```

Example - (Cont'd)

```
int main(){
    Shape *ptr;         // declare a pointer to base class
    Shape myshape;      // create objects
    Triangle t;
    Rectangle s;
    Circle c;

    ptr = &myshape;
    ptr -> showArea();

    ptr = &t;
    ptr -> setDim(10.0, 5.0);
    ptr -> showArea();

    ptr = &s;
    ptr -> setDim(10.0, 10.0);
    ptr -> showArea();

    ptr = &c;
    ptr -> setDim(10.0);
    ptr -> showArea();
  }
```

Pure Virtual Function

- A pure virtual function is a virtual function in base class that has no definition.
- E.g. Consider the virtual function showArea() in base class Shape; it has only an abstract meaning.Thus, showArea() can be declared as pure virtual function.
- A pure virtual function is declared using specifier “= 0 ”.

Abstract Classes

- A class that has a pure virtual function is an abstract class. Abstract class is used as an interface for its derived classes.
- If a class derived from an abstract class, and this class doesn’t override all the pure virtual funtion in the base class, then this class is also an abstract class.
- No object can be created for an abstract class !

Example

- Since class **Shape** has a pure virtual function, it becomes an abstract base class (ABC). Now, class **Triangle** must override **showArea()**.

```
class Shape {
public:
  void setDim(double, double = 0);
  virtual void showArea() = 0;      // pure
protected:
  double x, y;
};

class Triangle : public Shape{
  // must override Shape::showArea()
  // ...
};
```

Example - (Cont'd)

```
class Shape {              // Abstract base class
public:
  void setDim(double, double = 0);
  virtual void showArea() = 0;
protected:
  double x, y;
};
void Shape::setDim(double xx, double yy){
   x = xx;
   y = yy;
}

class Triangle : public Shape{
public:
   virtual void showArea();          // a must !
};
void Triangle::showArea(){
   cout << "Triangle with height " << x << " and base " << y
        << " has an area of " << x * y* 0.5 << "\n";
}
```

Example - (Cont'd)

```
class Rectangle : public Shape{
public:
   virtual void showArea();          // a must !
};
void Rectangle::showArea(){
    cout << "Rectangle with dimentions " << x << " and " << y
         << " has an area of " << x * y << "\n";
}
// ... class Circle
```

Example - (Cont'd)

```
int main(){
   Shape *ptr;      // pointers to ABC is ok.

   Shape myshape;   // wrong !
   Triangle t;
   Rectangle s;
   Circle c;

   ptr = &t;
   ptr -> setDim(10.0, 5.0);
   ptr -> showArea();

   ptr = &s;
   ptr -> setDim(10.0, 10.0);
   ptr -> showArea();

   ptr = &c;
   ptr -> setDim(10.0);
   ptr -> showArea();
 }
```