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.

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'';
}
```

In C, only compile-time binding is provided.

In C++, all non-virtual functions are bound at compile-time.

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.

A typical scenario of polymorphism in C++:

- There is an inheritance hierarchy
- The first class that defines a virtual function is the base class of the hierarchy that uses dynamic binding for that function name and signature.
- Each of the drived classes in the hierarchy must have a virtual function with same name and signature.
- There is a pointer of base class type; and this pointer is used to invoke virtual functions of derived class.

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 can not be stand-alone functions or static methods.

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

  Example:

  ```
  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.

  Example:

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

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

In order for a derived class virtual function instance to override the base class virtual function instance, its signature must match the base class virtual function exactly.

The overriding functions are virtual automatically. The use of keyword `virtual` is optional in derived classes.

Example:

```
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,

Example:

```
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 );
}
```

Thus, polymorphism in C++ is supported by using pointers and references.

Exercise 1:

Is `m()` bound at compile time or run time ?
Output ?

```
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 ?
Output?

```
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();
}
```

Name Hiding

When derived class adds a method with the same name but different signature to the base class virtual function, this new method hides the base class virtual function.

Example:

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

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

int main(){
   D d;
   d.m(5);   // Error! B::m(int) is hidden.
}
```

Exercise 3:

Is `m()` bound at compile time or run time ?

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

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

int main(){
  B *p;
  p = new D;
  p -> 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.
Example:

```
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();
}
```

Virtual Tables

C++ uses the virtual table (vtable) machanism to implement the dynamic binding of virtual functions.

- A class with virtual member functions has a virtual table which contains the address of its virtual functions.
- An object of such a class has a pointer(**`vptr`**) to point to the virtual table of the class.
- Dynamic binding is done by looking up the virtual table for the entry point of the appropriate function at run-time.

Example:

```
class B {
public:
   virtual void m1(){
      // ...
   }
   virtual void m2(){
      // ...
   }
};
class D :: B {
   void m1(){     // overide B::m1()
      // ...
   }
};

int main(){
   B *p;
   B b;
   D d;
   p = &d;     // p is set to d's  address
   p -> m1();
   p -> m2();
   p = &b;     // p is set to b's  address
   p -> m1()
   p -> m2();
}
```

Constructors and Destructors

- A constructor cannot be virtual since it is used to construct an object.
- A destructor can be virtual. Virtual destructors are very useful when some derived classes have cleanup code.

Example:

```
class B {
public:
   virtual B();         // error
   virtual ~B();        // ok
   virtual void f();    // ok
};
```

Example:

```
class B{
 public:
  B(){
    cout <<"constructing B. \n";
    bp = new char[5];
  }
  ~B(){
    cout <<"destructing B. \n";
    delete[] bp;
  }
 private:
  char *bp;
};
class D : public B{
 public:
  D(){
    cout <<"constructing D. \n";
    dp = new char[5000];
  }
  ~D(){
    cout <<"destructing D. \n";
    delete[] dp;
  }
 private:
  char *dp;
};
int main(){
  B *ptr = new D();
  delete ptr;
}
OUTPUT?
```

Fix the problem by using a virtual destructor.

```
class B{
public:
  // ...
  virtual ~B(){
    cout <<"destructing B. \n";
    delete[] bp;
  }
  // ...
};
class D : public B{
  // ...
};
int main(){
  B *ptr = new D();
  delete ptr;
}
```

When the destructor of base class is made virtual, destructors of derived classes are virtual automatically. Thus, run-time binding is in effect.

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
  - Architectural simplicity
- Compared to compile time binding, run time binding has overhead in terms of space and time.
  - Extra space is needed for virtual table.
  - Extra time for virtual table lookup is required at each polymorfic function call.

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";
}
```

In base class `Shape`:

- `setDim()` is a non-virtual member function since its operation is common to all derived classes.
- `showArea()` is declared virtual since the area of each object is computed differently.

```
// 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;
}
```

```
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 ".

Note:

- Only a virtual member function can be pure.

```
void f() = 0;        //error! f() is a stand alone function

class B{
public:
   void setX() = 0;  //error! setX  not virtual
   // ...
};
```

- Declaring a virtual function pure is not the same as defining a virtual function with an empty body.

```
class B{
public:
   virtual void setX(){}   // virtual but not pure
   // ...
};
```

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 !

Therefore, classes derived from an abstract class must override all of the base class's pure virtual functions to become "non-abstract".

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";
}

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
```

```
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();
 }
```

Virtual Multiple Inheritance

```
class A { ... };

class B : public A { ... };
class C : public A { ... };

class D : public B, public C { ... };
```

v.s.

```
class A { ... };

class B : virtual public A { ... };
class C : virtual public A { ... };

class D : public B, public C { ... };
```

Run-Time Type Checking

C++ supports run-time type identification(RTTI). It provides mechanisms to

- Check type conversion at run time.
- Determine the actual derived object's type that a pointer(or reference) refers to at run time.

Two operators are provided for RTTI support:

- `dynamic_cast` operator
- `typeid` operator

Used only for polymorphic types, e.g. types with virtual-functions.

A pointer of derived class can be assigned to a pointer of base class, which is known as **upcast**.
A pointer of base class can not be assigned to a pointer of derived class, which is known as **downcast**.
Example:

```
class B{
 public:
  int zip() { return x; }
 private:
  int x;
};
class D : public B{
 public:
  int zap() { return y; }
 private:
  int y;
};
int main(){
  D *dptr;
  B *bptr;
  bptr = dptr;   // ok, upcast
  dptr = bptr;   // compile time error
                          // Cannot assign B* to D* downcast
  dptr = static_cast< D* >( bptr );  // ?
}
```

Downcast might not be safe, but can not be detected by the compiler.

`static_cast` is not type-safe. Run-time error may occur.

Example:

```
int main(){
  D *dptr;
  B *bptr = new B;
  dptr = static_cast< D* >( bptr );
  dptr -> zap();  // ?
}
```

C++ provides **dynamic_cast** for safe type conversion.

```
class B{
 public:
  virtual int zip() { return x; }
 private:
  int x;
};
class D : public B{
  // ...
};
int main(){
   D *dptr;
   B *bptr = new B;
   dptr = dynamic_cast< D* >( bptr );
   if (dptr)       // check if cast is successful
     dptr -> zap();
   else
     cerr << "Cast not safe \n";
 }
```

dynamic_cast is legal only on a polymorphic type.

dynamic_cast performs two operations at once. First, it verifies that the cast is valid. Then only if the cast is valid does it perform the cast, otherwise it returns a null pointer.

`dynamic_cast` provides an alternative to the virtual function machanism.

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

class Employee{
 public:
  virtual void salary() { cout << "Employee::salary"; }
};
class Manager : public Employee{
 public:
  void salary() { cout << "Manager::salary"; }
};
class Programmer : public Employee{
 public:
  void salary() { cout << "Programmer::salary"; }
  void bonus() { cout << "Programmer::bonus"; }
};
void paycheck( Employee *ep ){
  Programmer *pp = dynamic_cast< Programmer* >( ep );
  if (pp)
    pp -> bonus();
  else
    ep -> salary();
}
int main(){
  Employee *eptr = new Programmer;
  paycheck( eptr );
  eptr = new Manager;
  paycheck( eptr );
}
```

C++ also provides a **`typeid`** operator that allows queries of type information at run-time.

Example:

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

class B_class{
public:
  virtual void f(){}
  // ...
};
class D_class : public B_class{
  // ...
};

int main(){
  int x;
  cout << typeid( x ).name() << endl;
  cout << typeid( 8.16 ).name() <<endl;

  D_class dobj;
  B_class *bptr = &dobj;

  cout << typeid( bptr ).name() << endl;
  cout << typeid( *bptr ).name() << endl;
}
```

The result of the **`typeid`** operator can be compared.

Example:

```
B_class *bptr = new D_class;

typeid( bptr ) == typeid( B_class* )    // true
typeid( bptr ) == typeid( D_class* )    // false
typeid( bptr ) == typeid( B_class )     // false
typeid( bptr ) == typeid( B_class )     // false

typeid( *bptr ) == typeid( D_class )    // true
typeid( *bptr ) == typeid( B_class )    // false
```

The **`typeid`** operator is used for advanced system programming.