Module 27

Partha Pratim Das

Module 27: Programming in C++

Dynamic Binding (Polymorphism): Part 2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

Module Objectives

- Understand Static and Dynamic Binding
- Understand Polymorphic Type

Module Outline

- Binding
  - Types
  - Static Binding
  - Dynamic Binding
- Polymorphic Type

Type of an Object

- The static type of the object is the type declared for the object while writing the code
- Compiler sees static type
- The dynamic type of the object is determined by the type of the object to which it currently refers
- Compiler does not see dynamic type

```
class A {};
class B : public A {};

int main() {
    A *p;
    p = new B; // Static type of p = A
               // Dynamic type of p = B
}
```

Static and Dynamic Binding

- **Static binding** (early binding): When a function invocation binds to the function definition based on the static type of objects
- This is done at compile-time
- Normal function calls, overloaded function calls, and overloaded operators are examples of static binding
- **Dynamic binding** (late binding): When a function invocation binds to the function definition based on the dynamic type of objects
- This is done at run-time
- Function pointers, Virtual functions are examples of late binding

Static Binding

Inherited Method	Overridden Method

`#include<iostream>`
`using namespace std;`
`class B { public:`
`    void f() {}`
`};`
`class D : public B { public:`
`    void g() {} // new function`
`};`
`int main() {`
`    B b;`
`    D d;`
 	
`    b.f(); // B::f()`
`    d.f(); // B::f() ----- Inherited`
`    d.g(); // D::g() ----- Added`
`}`
	`#include<iostream>`
	`using namespace std;`
	`class B { public:`
	`    void f() { }`
	`};`
	`class D : public B { public:`
	`    void f() { }`
	`};`
	`int main() {`
	`    B b;`
	`    D d;`
	
	`    b.f(); // B::f()`
	`    d.f(); // D::f() ----- Overridden`
	`           // masks the base class function`
	`}`

• Object `d` of derived class inherits the base class function `f()` and has its own function `g()` • Function calls are resolved at compile time based on static type	• If a member function of a base class is redefined in a derived class with the same signature then it masks the base class method • The derived class method `f()` is linked to the object `d`. As `f()` is redefined in the derived class, the base class version cannot be called with the object of a derived class

Member Functions – Overrides and Overloads: RECAP (Module 22)

Inheritance	Override & Overload

Inheritance

```
class B { // Base Class
public:
    void f(int i);
    void g(int i);
};
class D: public B { // Derived Class
public:
    // Inherits B::f(int)
    // Inherits B::g(int)




};

B b;
D d;

b.f(1); // Calls B::f(int)
b.g(2); // Calls B::g(int)

d.f(3); // Calls B::f(int)
d.g(4); // Calls B::g(int)
```

Override & Overload

```
class B { // Base Class
public:
    void f(int);
    void g(int i);
};
class D: public B { // Derived Class
public:
    // Inherits B::f(int)
    void f(int);      // Overrides B::f(int)
    void f(string&); // Overloads B::f(int)
    // Inherits B::g(int)
    void h(int i);    // Adds D::h(int)
};

B b;
D d;

b.f(1);      // Calls B::f(int)
b.g(2);      // Calls B::g(int)

d.f(3);      // Calls D::f(int)
d.g(4);      // Calls B::g(int)

d.f("red"); // Calls D::f(string&)
d.h(5);      // Calls D::h(int)
```

- D::f(int) overrides B::f(int)
- D::f(string) overloads B::f(int)

using Construct – Avoid Method Hiding

```
#include<iostream>
using namespace std;
class A { public:
    void f() {}
};

class B : public A {
     // To overload, rather than hide the base class function f()
     // is introduced into the scope of B with a using declaration
    using A::f;
    void f(int) { }
};

int main() {
    B b; // function calls resolved at compile time

    b.f(3); // B::f(int)
    b.f();  // A::f()
}
```

- Object `b` of derived class linked to with inherited base class function `f()` and the overloaded version defined by the derived class `f(int)`, based on the input parameters – function calls resolved at compile time

Dynamic Binding

Non-Virtual Method	Virtual Method

<pre>#include<iostream>
using namespace std;
class B { public:
    void f() { }
};
class D : public B { public:
    void f() { }
};
int main() {
    B b;
    D d;
 	
    B *p;
 	
    p = &b; p->f(); // B::f()
    p = &d; p->f(); // B::f()
}</pre>
	<pre>#include<iostream>
	using namespace std;
	class B { public:
	    virtual void f() { }
	};
	class D : public B { public:
	    virtual void f() { }
	};
	int main() {
	    B b;
	    D d;
	
	    B *p;
	
	    p = &b; p->f(); // B::f()
	    p = &d; p->f(); // D::f()
	}</pre>

• `p->f()` always binds to `B::f()` • Binding is decided by the type of pointer • **Static Binding**	• `p->f()` binds to `B::f()` for a `B` object, and to `D::f()` for a `D` object • Binding is decided by the type of object • **Dynamic Binding**

Static and Dynamic Binding: RECAP (Module 26)

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

class B {
public:
    void f() { cout << "B::f()" << endl; }
    virtual void g() { cout << "B::g()" << endl; }
};

class D: public B {
public:
    void f() { cout << "D::f()" << endl; }
    virtual void g() { cout << "D::g()" << endl; }
};
```

```
int main() {
    B b;
    D d;

    B *pb = &b;
    B *pd = &d; // UPCAST

    B &rb = b;
    B &rd = d;  // UPCAST

    b.f(); // B::f()
    b.g(); // B::g()
    d.f(); // D::f()
    d.g(); // D::g()
```

```
    pb->f(); // B::f() -- Static Binding
    pb->g(); // B::g() -- Dynamic Binding
    pd->f(); // B::f() -- Static Binding
    pd->g(); // D::g() -- Dynamic Binding

    rb.f();  // B::f() -- Static Binding
    rb.g();  // B::g() -- Dynamic Binding
    rd.f();  // B::f() -- Static Binding
    rd.g();  // D::g() -- Dynamic Binding

    return 0;
}
```

Polymorphic Type: Virtual Functions

- Dynamic binding is possible only for pointer and reference data types and for member functions that are declared as virtual in the base class.
- These are called Virtual Functions
- If a member function is declared as virtual, it can be overridden in the derived class
- If a member function is not virtual and it is re-defined in the derived class then the latter definition hides the former one
- Any class containing a virtual member function – by definition or by inheritance – is called a Polymorphic Type
- A hierarchy may be polymorphic or non-polymorphic
- A non-polymorphic hierarchy has little value

Polymorphism Rule

```
#include <iostream>
using namespace std;
class A { public:
    void f()          { cout << "A::f()" << endl; } // Non-Virtual
    virtual void g()  { cout << "A::g()" << endl; } // Virtual
    void h()          { cout << "A::h()" << endl; } // Non-Virtual
};
class B : public A { public:
    void f()          { cout << "B::f()" << endl; } // Non-Virtual
    void g()          { cout << "B::g()" << endl; } // Virtual
    virtual void h()  { cout << "B::h()" << endl; } // Virtual
};
class C : public B { public:
    void f()          { cout << "C::f()" << endl; } // Non-Virtual
    void g()          { cout << "C::g()" << endl; } // Virtual
    void h()          { cout << "C::h()" << endl; } // Virtual
};
```

```
int main() { B *q = new C; A *p = q;

    p->f();
    p->g();
    p->h();

    q->f();
    q->g();
    q->h();

    return 0;
}
```

```
A::f()
C::g()
A::h()
B::f()
C::g()
C::h()
```

Module Summary

- Static and Dynamic Binding are discussed in depth
- Polymorphic type introduced

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655