



COMP 345 Fall 18

Week 8

Haotao Lai (Eric)
h_lai@encs.concordia.ca



Lab Instructor

Section: B-X 9999 --W---- 20:30 22:20 H929

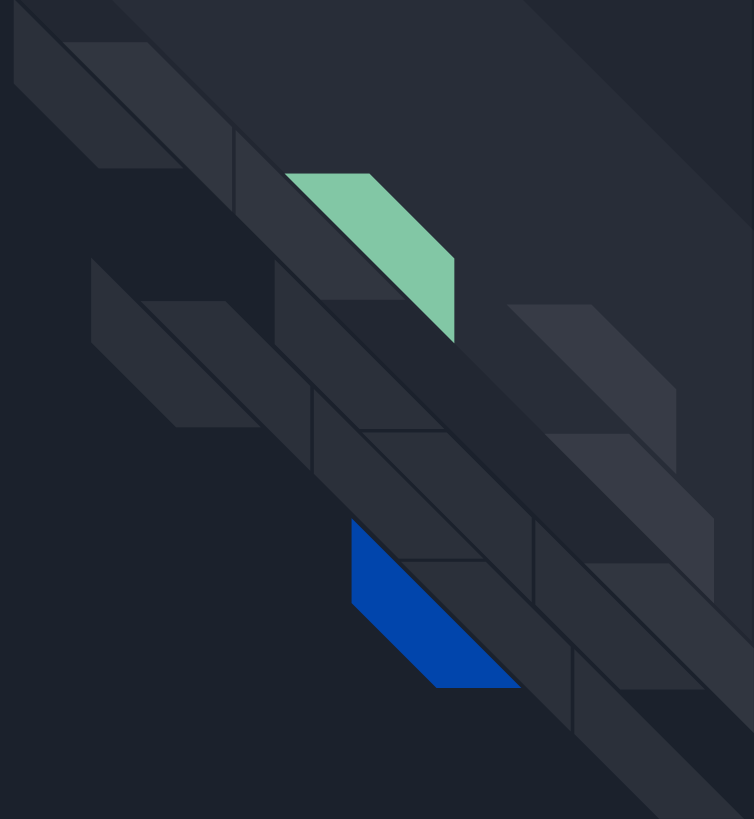
Name: Haotao Lai (Eric)

Office: EV 8.241

Email: h_lai@encs.concordia

Website: <http://laihaotao.me/ta>

Polymorphism





Static dispatch vs. Dynamic dispatch

Static dispatch is a form of polymorphism fully resolved during **compile time**.

Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at **run time**.

Question

method overloading and method overriding, which belong to static dispatch and which belong to dynamic dispatch? why?



An Java version example

```
// overload example
class OverloadExample {
    public int foo(int k) {
        return k;
    }
    public int foo(int i, int j) {
        return i + j;
    }
}

// override example
class OverrideExampleBase {
    public int foo(int i, int j) {
        return i + j;
    }
}

class OverrideExampleDerived extends OverrideExampleBase {
    public int foo(int i, int j) {
        return i * j;
    }
}

// possible consumer code
class Driver {
    public static void main() {
        OverloadExample e1 = new OverloadExample();
        e1.foo(10); // result 10
        e1.foo(10, 20); // result 30

        OverrideExampleBase e2 = new OverrideExampleBase();
        OverrideExampleBase e3 = new OverrideExampleDerived();
        e2.foo(10, 20); // result 30
        e3.foo(10, 20); // result 200
    }
}
```



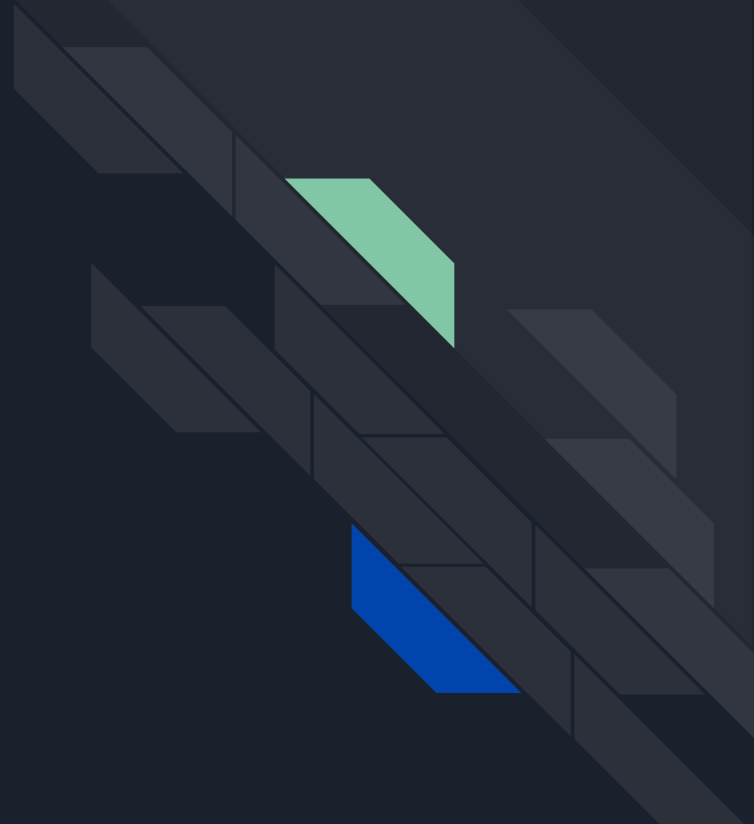
Single Inheritance

For Java, it means using the keyword “extends”.

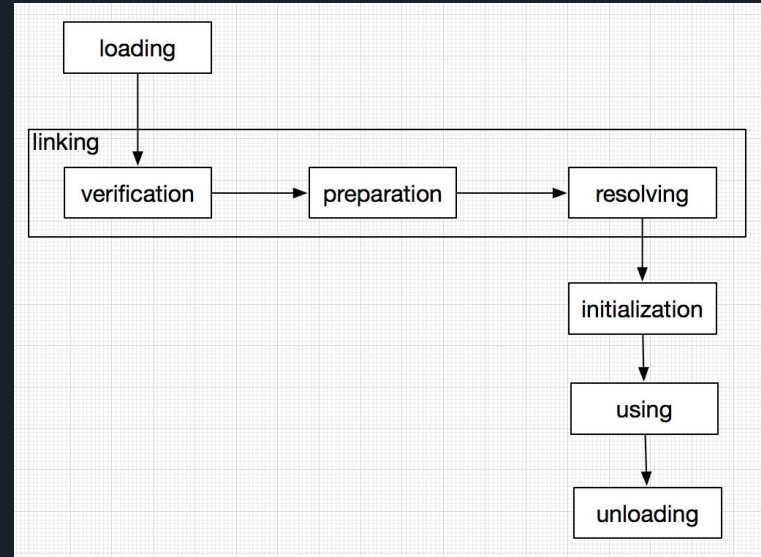
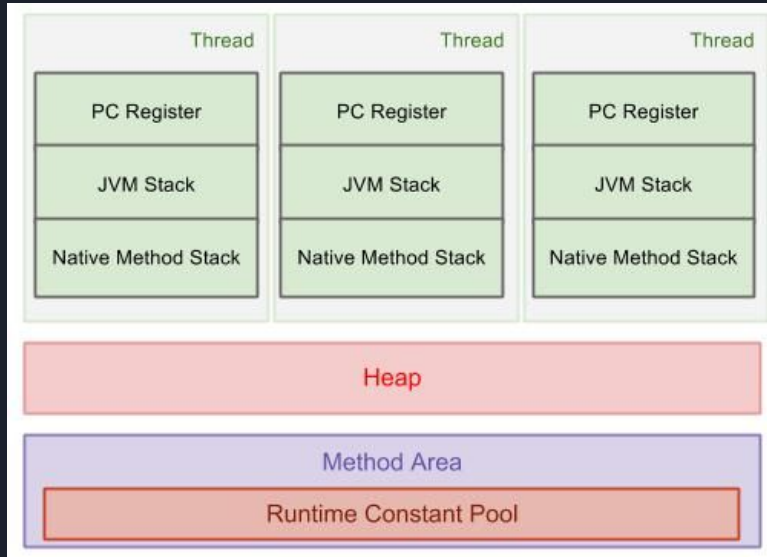
For C++, it means a subclass only has one direct superclass.

For single inheritance, they all use offset to figure out which method should be invoked.

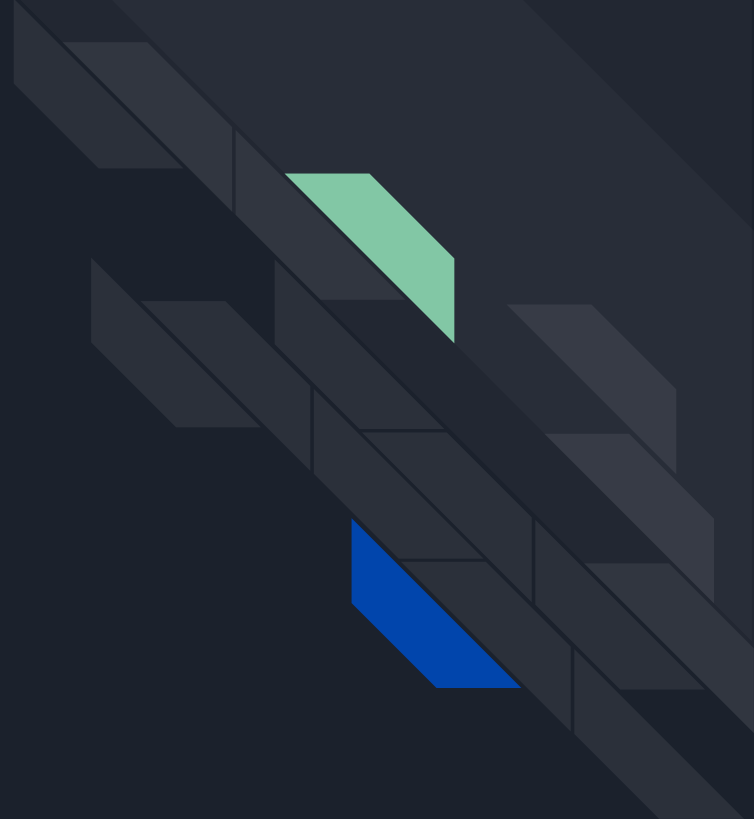
Question?



Java Runtime



Java Implementation




```
class Base {
    public String toString() {
        return "override toString in Base";
    }
    public void foo() {
        System.out.println("foo method in Base");
    }
}

class Derived1 extends Base {
    public void foo() {
        System.out.println("foo method in Derived1");
    }
}

class Derived2 extends Base {
    public void foo() {
        System.out.println("foo method in Derived2");
    }
}
```

```
// possible consumer code
class Driver {
    public static void main() {
        Base ref = new Base();
        Base ref1 = new Derived1();
        Base ref2 = new Derived2();

        ref.equals(ref1); // false, equals() method inherits from Object
        ref1.toString(); // "override toString in Base"
        ref2.toString(); // "override toString in Base"
        ref1.foo(); // "foo method in Derived1"
        ref2.foo(); // "foo method in Derived2"
    }
}
```



```
// possible consumer code
class Driver {
    public static void main() {
        Base ref = new Base();
        Base ref1 = new Derived1();
        Base ref2 = new Derived2();

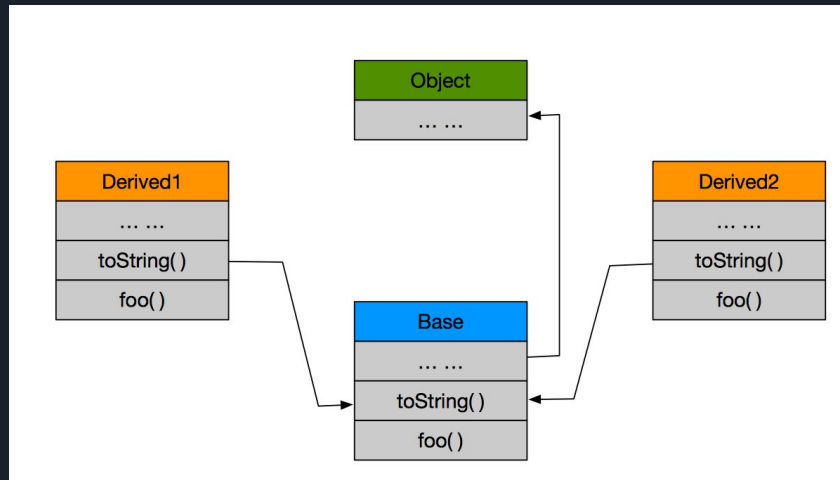
        ref.equals(ref1); // false, equals() method inherits from Object
        ref1.toString(); // "override toString in Base"
        ref2.toString(); // "override toString in Base"
        ref1.foo(); // "foo method in Derived1"
        ref2.foo(); // "foo method in Derived2"
    }
}
```

The problem is that in the compilation time, the compiler only know the type of “ref2” is Base, instead of Derived2.

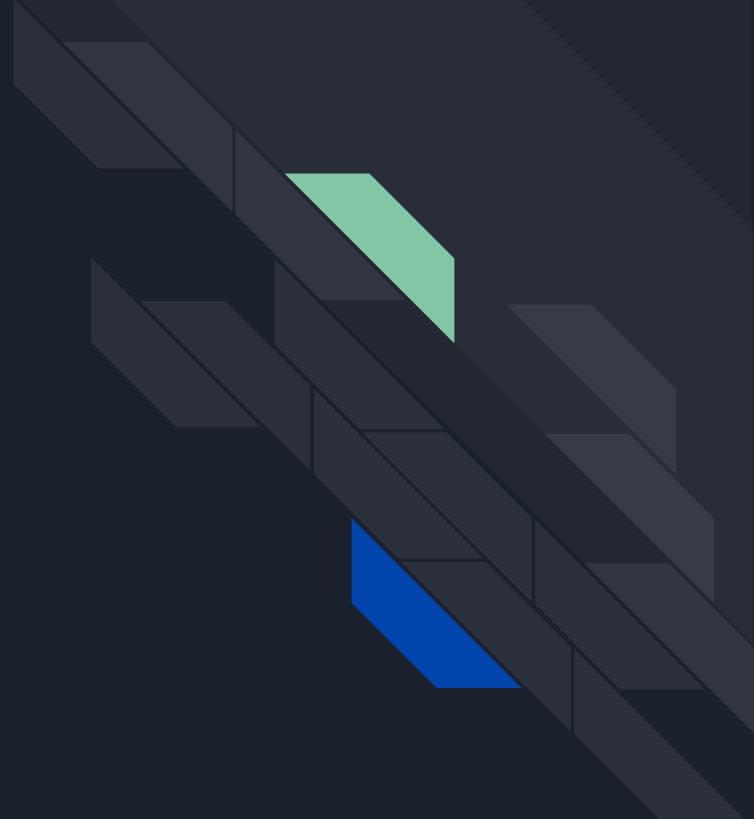


In the memory model, a class's method list always copy it superclass's inheritable methods list first and then followed by the subclass only method.

So we can use **offset** to calculate the invoking method address even we only know its superclass.



C++ Implementation

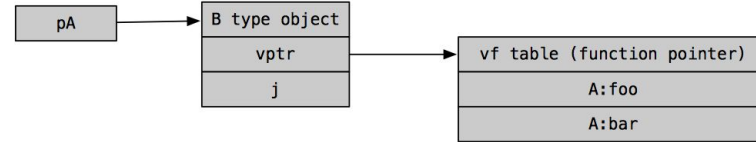


Virtual function table

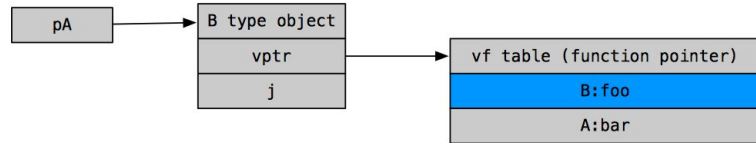
```
class A {  
    int i;  
    virtual void foo();  
    virtual void bar();  
};
```

```
class B : public A {  
    int j;  
    virtual void foo();  
    virtual void bbb();  
};
```

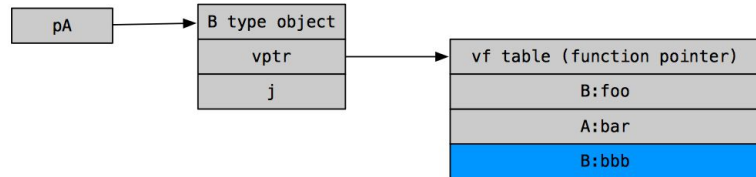
```
int main() {  
    A *pA = new B();  
    pA->foo();  
    return 0;  
}
```



step1: copy the virtual function table from its parent



step2: modify the pointer for overrided function



step3: add the new defined virtual function



Multiple Inheritance / Implement interfaces

For Java, it means using keyword “implements”. The Implementation for that is kind of brute force searching, it goes through the whole method list and found the method with exactly the same signature.

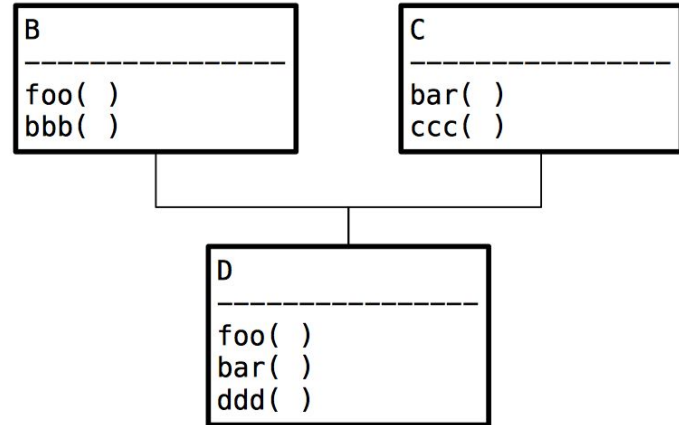
For C++, since it doesn't have such a powerful runtime system. It delegate more works to the compiler, still using the virtual function table.

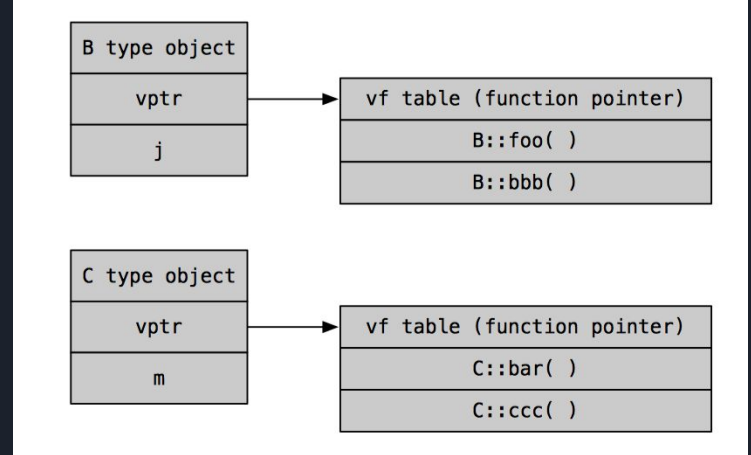
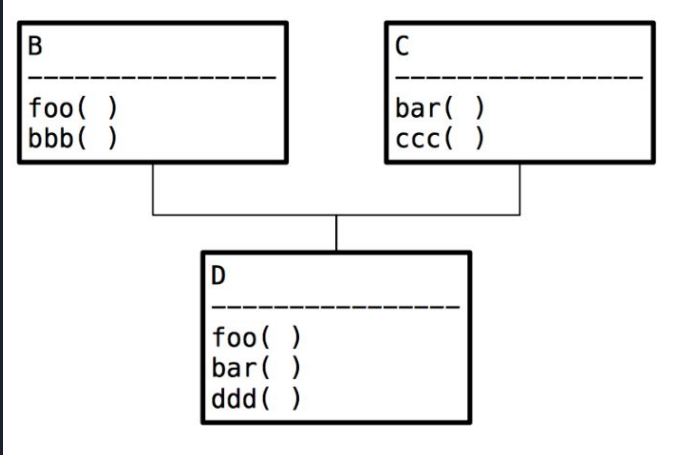
```
class B {
public:
    int j;
    virtual void foo();
    virtual void bbb();
};
```

```
class C {
public:
    int m;
    virtual void bar();
    virtual void ccc();
};
```

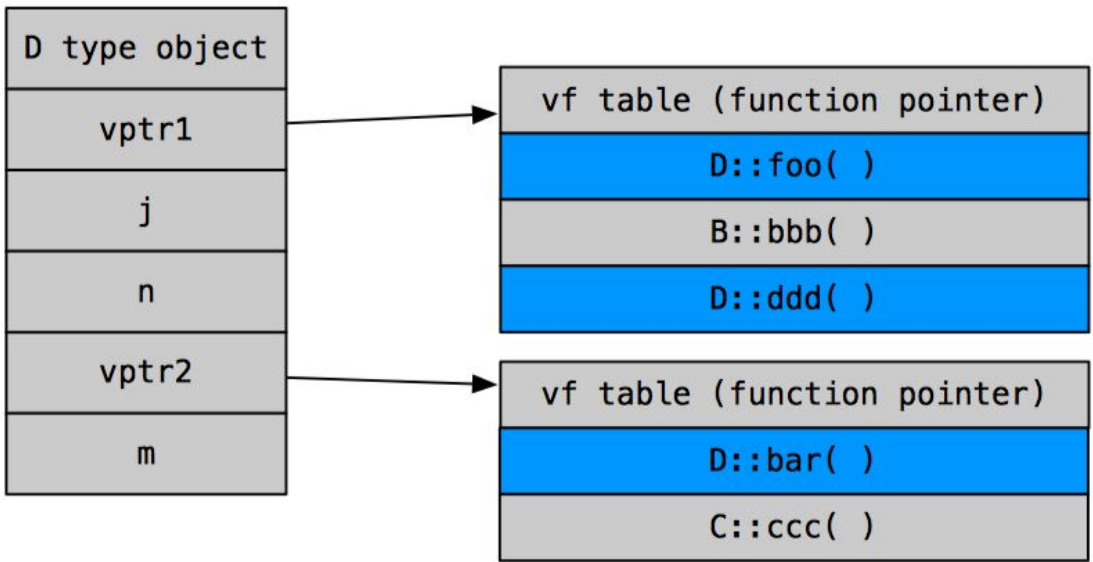
```
class D : public B, public C {
public:
    int n;
    virtual void foo();
    virtual void bar();
    virtual void ddd();
};
```

```
int main() {
    D *pD = new D();
    B *pB = pD;
    C *pC = pD;
}
```





How is the object D memory layout will look like?



```
class B {
public:
    int j;
    virtual void foo();
    virtual void bbb();
};
```

```
class C {
public:
    int m;
    virtual void bar();
    virtual void ccc();
};
```

```
class D : public B, public C {
public:
    int n;
    virtual void foo();
    virtual void bar();
    virtual void ddd();
};
```

```
int main() {
    D *pD = new D();
    B *pB = pD;
    C *pC = pD;
}
```

Question 1

How to make pC point to there?

```
D *tmp = new D();
C *pC = tmp? tmp + sizeof(B) : 0;
```

Question 2

What happen if we say delete pC;

Add offset for each virtual function table

