CS 247: Software Engineering Principles

Polymorphism

Reading: Eckel, Vol. 1
Ch. 15 Polymorphism & Virtual Functions

Review: Static vs. Dynamic Types

Given a pointer or reference to an object, there are two types to consider:

- static type of the pointer or reference.
- dynamic type of the referant (object).

```
class Base {
public:
  virtual void vfunc();
};

class Derived : public Base {
public:
  virtual void vfunc();
};

int main ()
{
   Base* bPtr = new Derived;
}
```

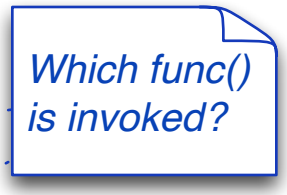
*static type is Base**

dynamic type is Derived

Review: Static vs. Dynamic Binding

```
int main (void)
{
   Base b;
   b.vfunc();

   Base *bPtr = new Derived;
   bPtr->vfunc();
}
```



1. Is method call legal?
Determined by the static type of pointer / reference

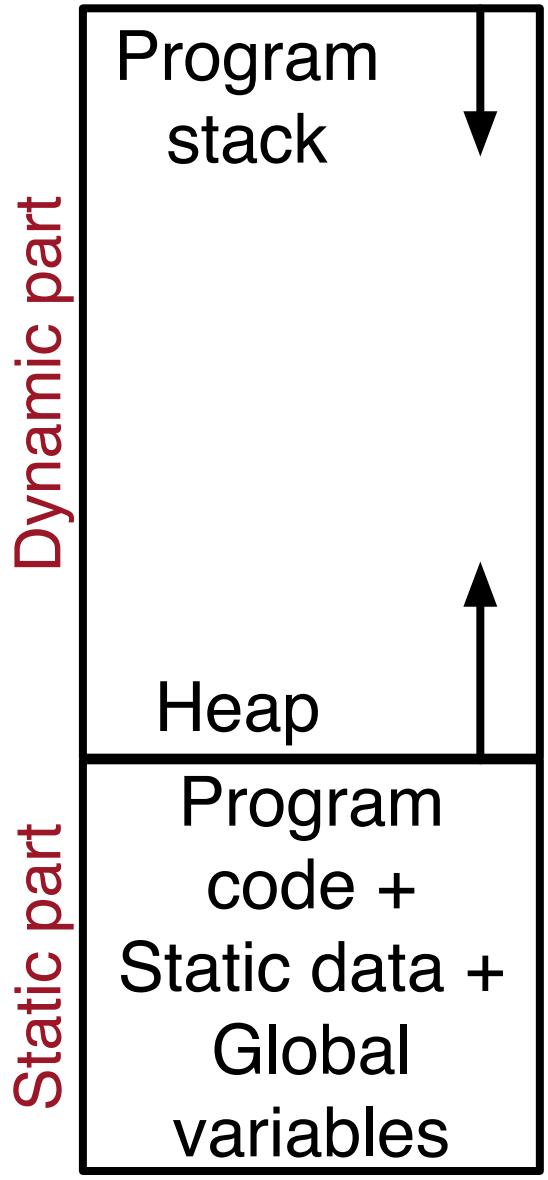
2. Which method is actually invoked?
Static binding - calls to non-virtual methods resolved at compile-time, based on the static type of the pointer / reference / object.

Dynamic binding - calls to virtual methods are resolved at run-time, based on the dynamic type of the referent of a pointer / reference.

Review: Run-Time Memory Model

- Dynamic part
  - program stack
  - heap

- Static part
  - program code
  - static variables
  - vtables
  - etc.

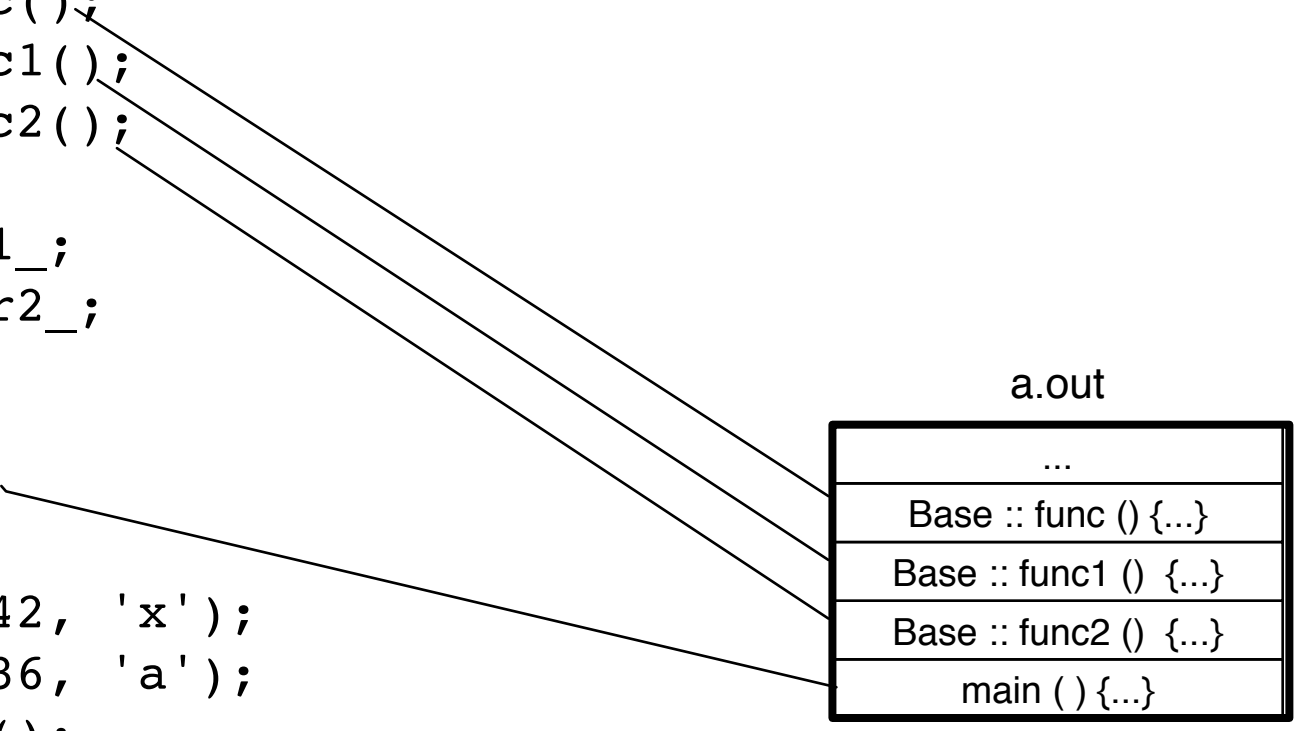


Program Code (compiled and linked)

a.out includes executable code for every function and method definition.

```
class Base {
public:
   ...
   void func();
   void func1();
   void func2();
private:
   int attr1_;
   char attr2_;
};

int main ()
{
   Base b1(42, 'x');
   Base b2(36, 'a');
   b1.func2();
}
```

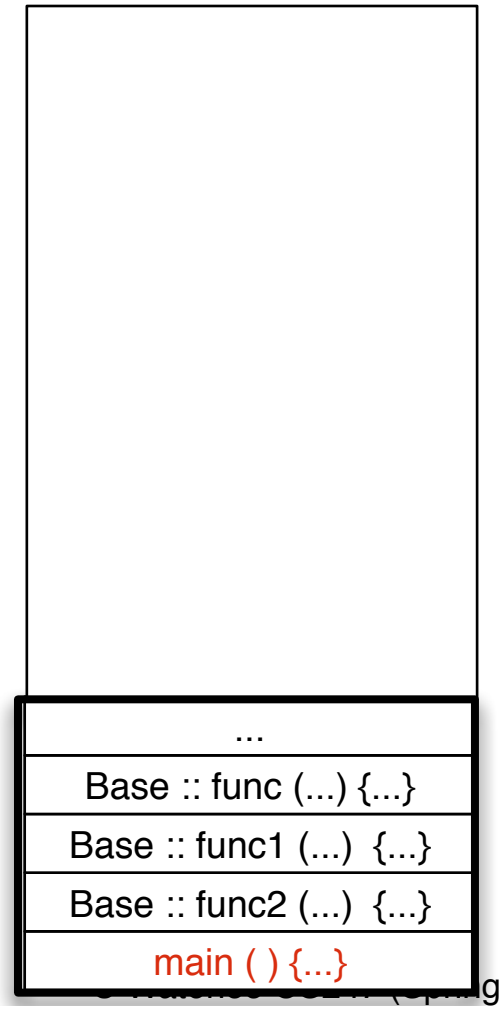


Static Binding

Compiler knows at compile-time which method is executed.

```
class Base {
public:
   ...
   void func();
   void func1();
   void func2();
private:
   int attr1_;
   char attr2_;
};

int main ()
{
   Base b1(42, 'x');
   Base b2(36, 'a');
   b1.func2();
}
```

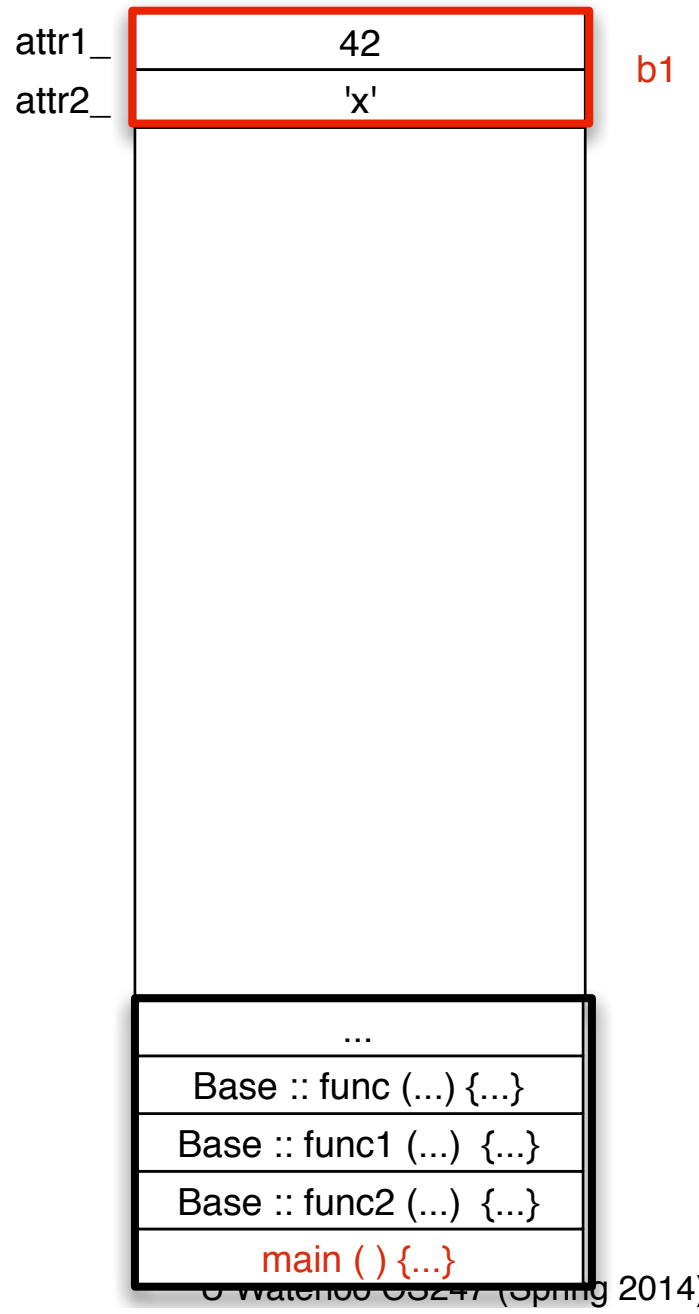


Object Construction

Object b1 is constructed.

```
class Base {
public:
   ...
   void func();
   void func1();
   void func2();
private:
   int attr1_;
   char attr2_;
};

int main (void)
{
   Base b1(42, 'x');
   Base b2(36, 'a');
   b1.func2();
}
```

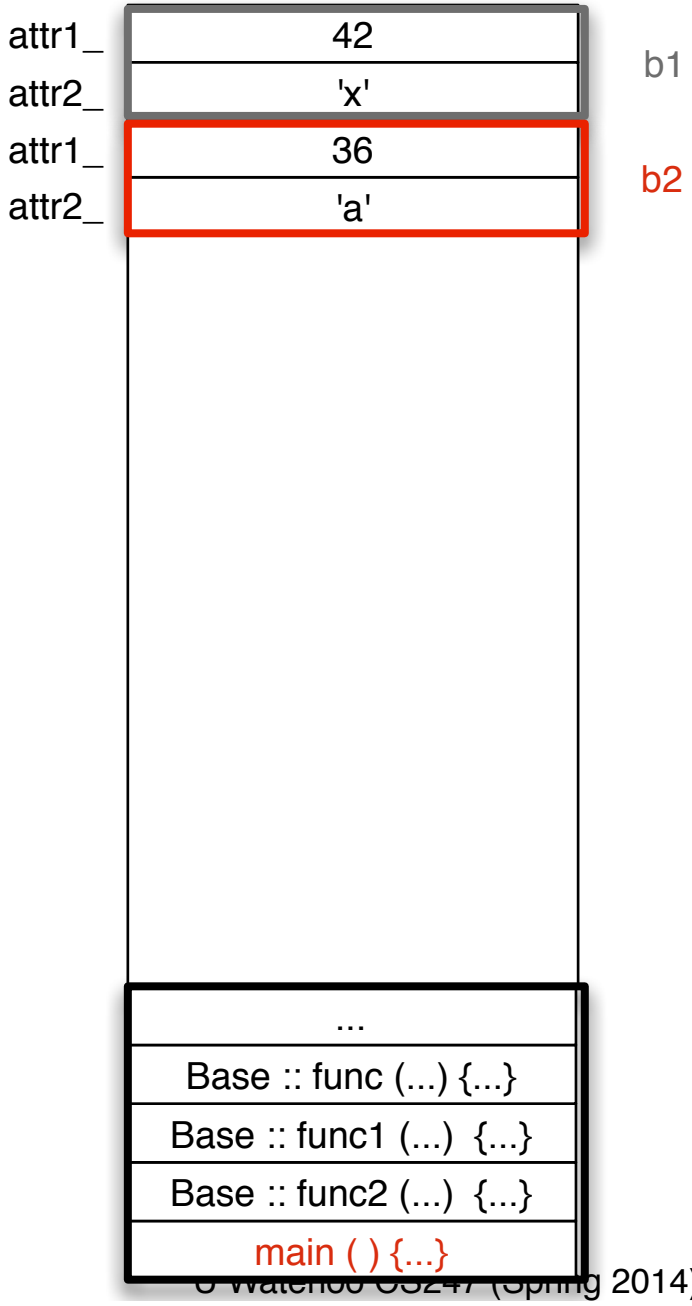


Object Construction

Object b2 is constructed.

```
class Base {
public:
   ...
   void func();
   void func1();
   void func2();
private:
   int attr1_;
   char attr2_;
};

int main (void)
{
   Base b1(42, 'x');
   Base b2(36, 'a');
   b1.func2();
}
```

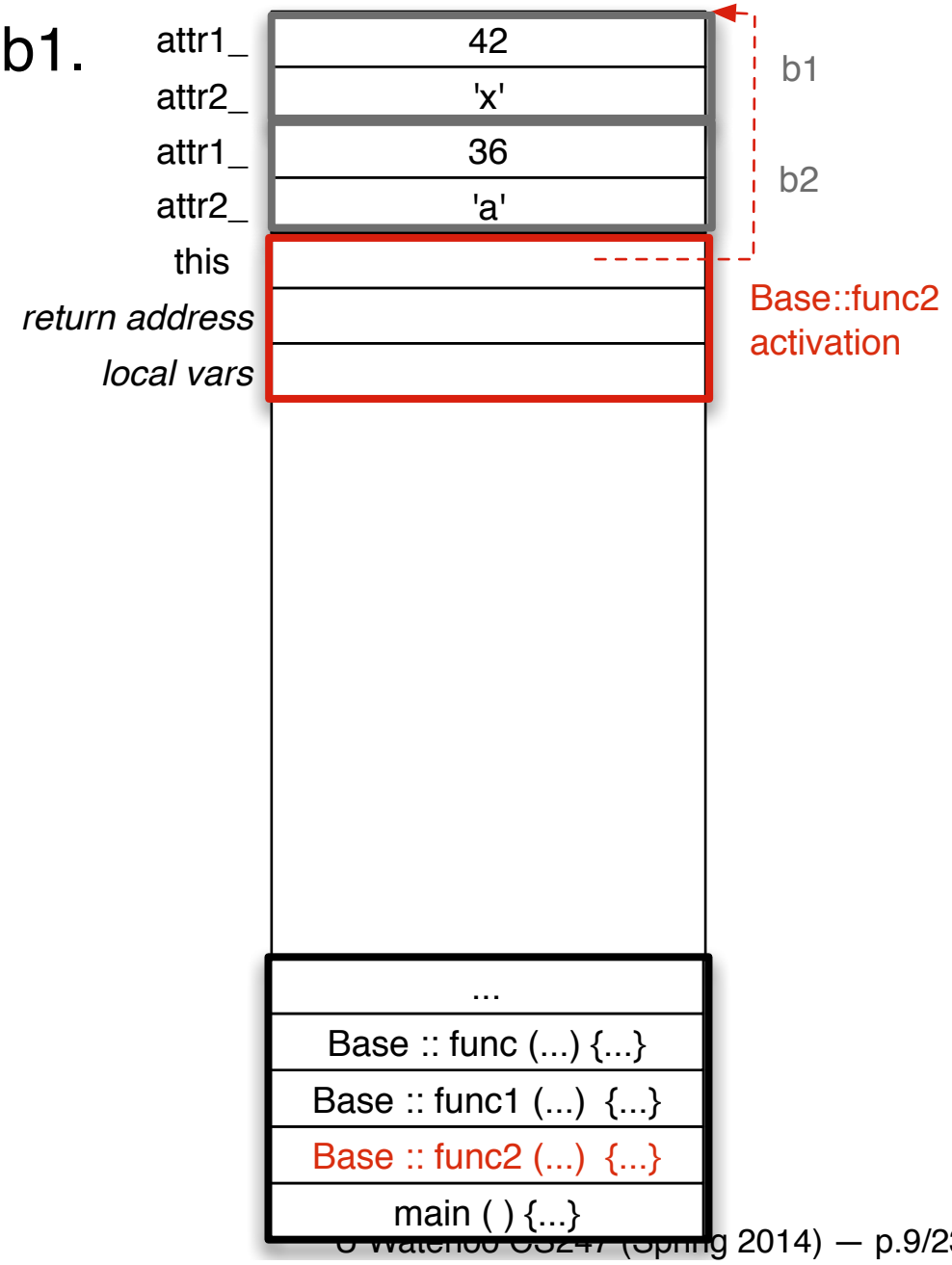


Method Invocation

func2() is invoked on object b1.

```
class Base {
public:
   ...
   void func();
   void func1();
   void func2();
private:
   int attr1_;
   char attr2_;
};

int main (void)
{
   Base b1(42, 'x');
   Base b2(36, 'a');
   b1.func2();
}
```

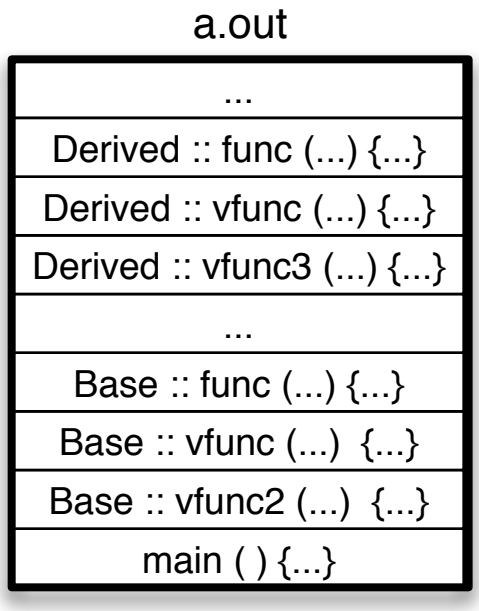


Let's Add Virtual Functions

```
class Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc2();
private:
   int attr1_;
   char attr2_;
};

class Derived : public Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc3();
private:
   float attr3_;
};

int main (void) {...}
```



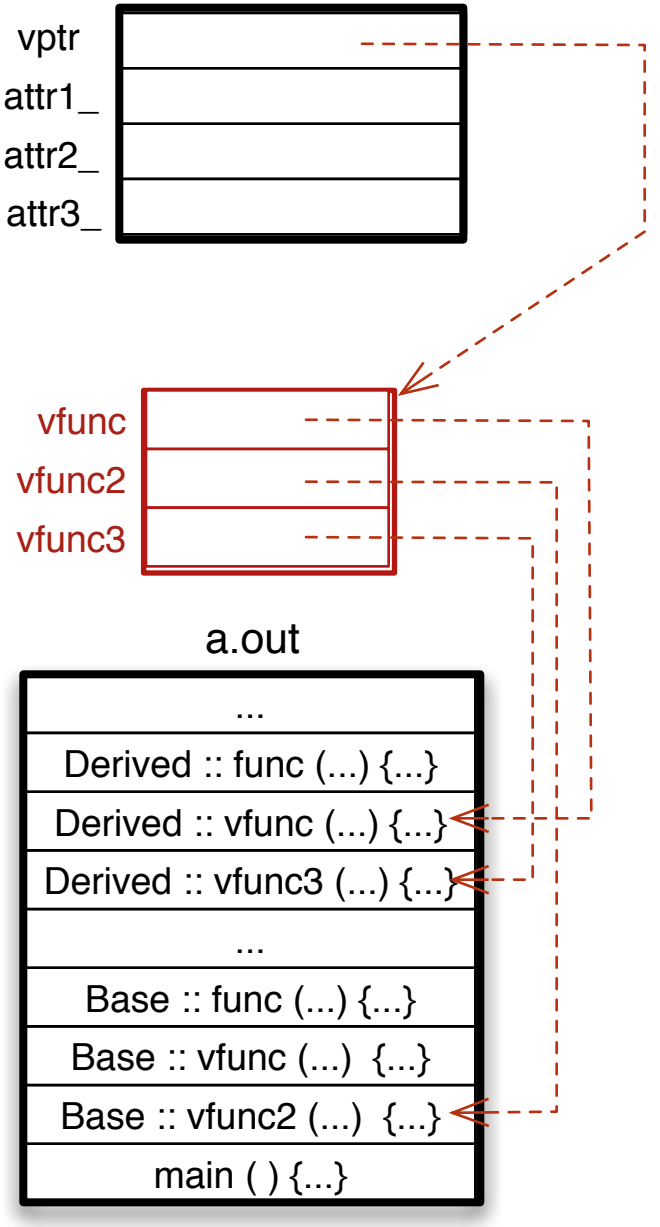
VTables

- Each class with virtual functions has a vtable of pointers to functions.
- Every object with virtual functions has a pointer to its class's vtable.

```
class Base {
public:
   ...

};

class Derived : public Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc3();
private:
   float attr3_;
};
```

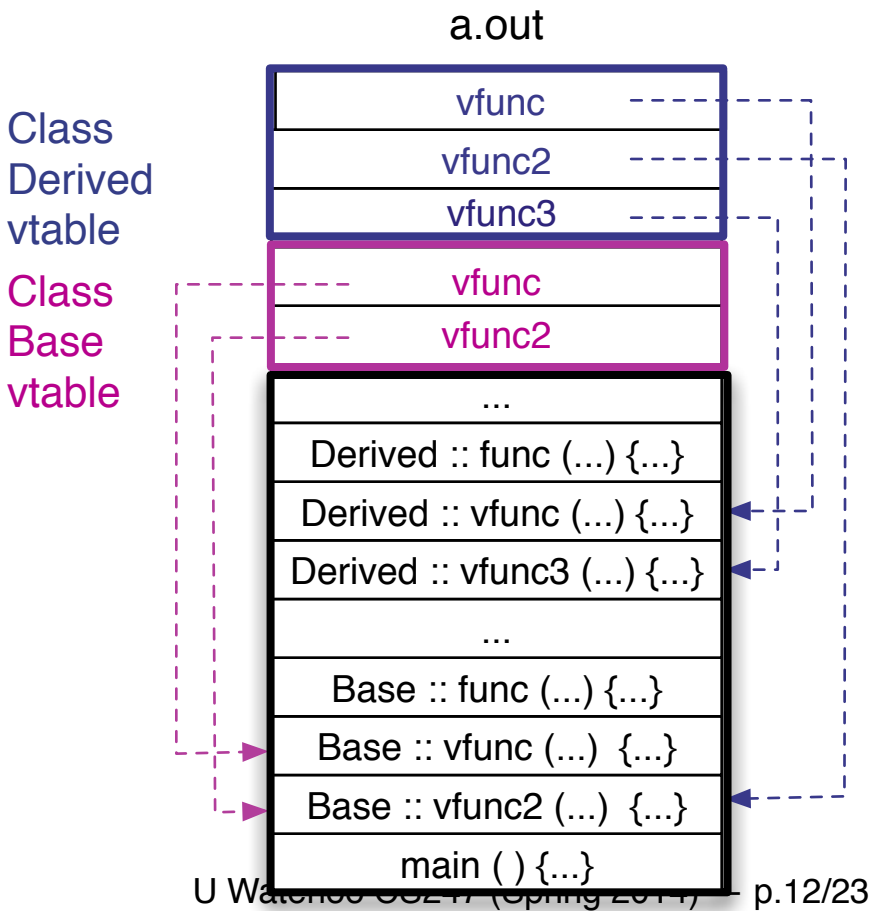


VTables

Memory map now includes vtables for both classes.

```
class Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc2();
private:
   int attr1_;
   char attr2_;
};

class Derived : public Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc3();
private:
   float attr3_;
};
```

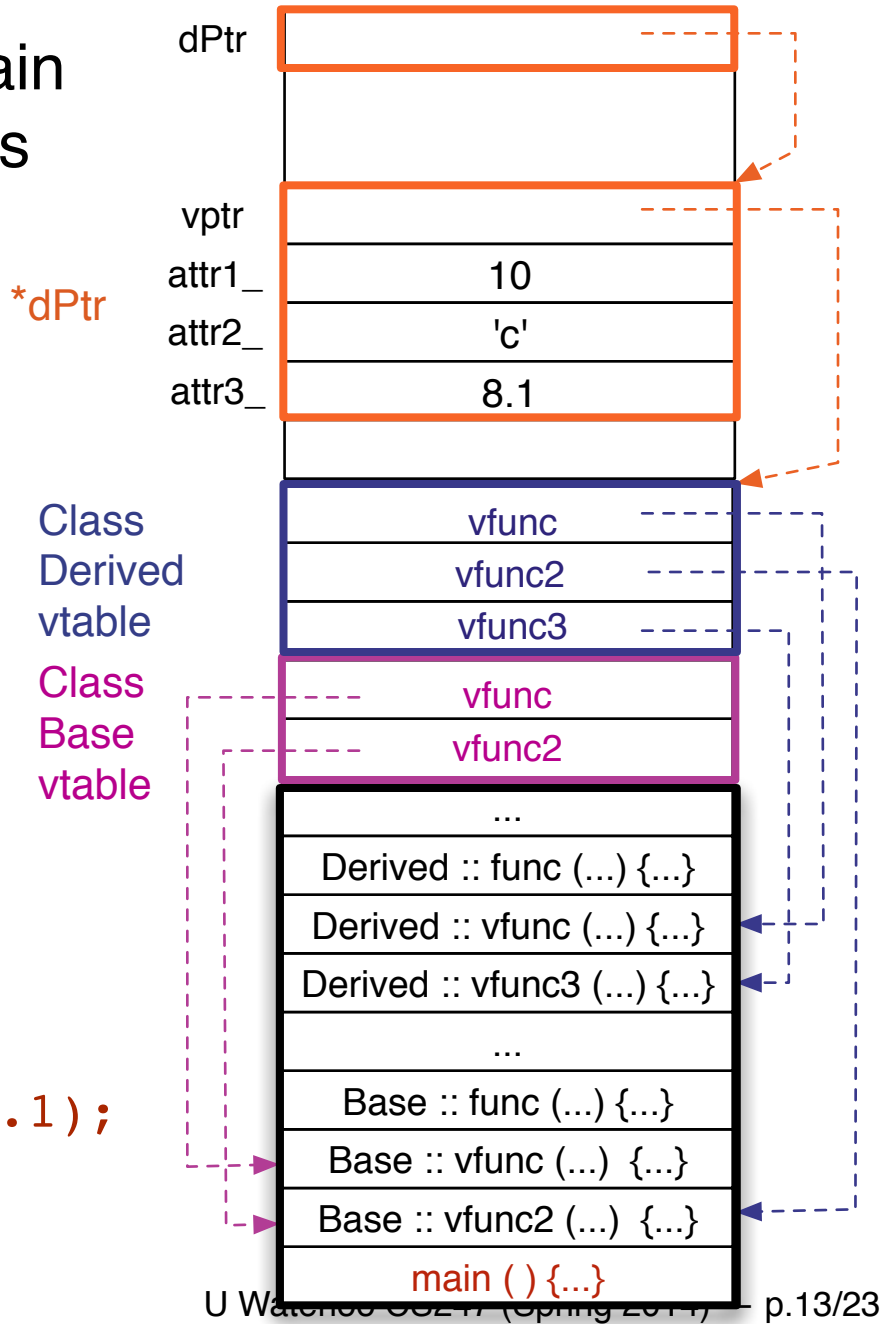


Object Construction

Note that in this example, the main function declares and instantiates pointer variables.

```
class Derived : public Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc3();
private:
   float attr3_;
};

int main (void) {
   Base *dPtr;
   dPtr = new Derived(10, 'c', 8.1);
   dPtr->vfunc2();
}
```

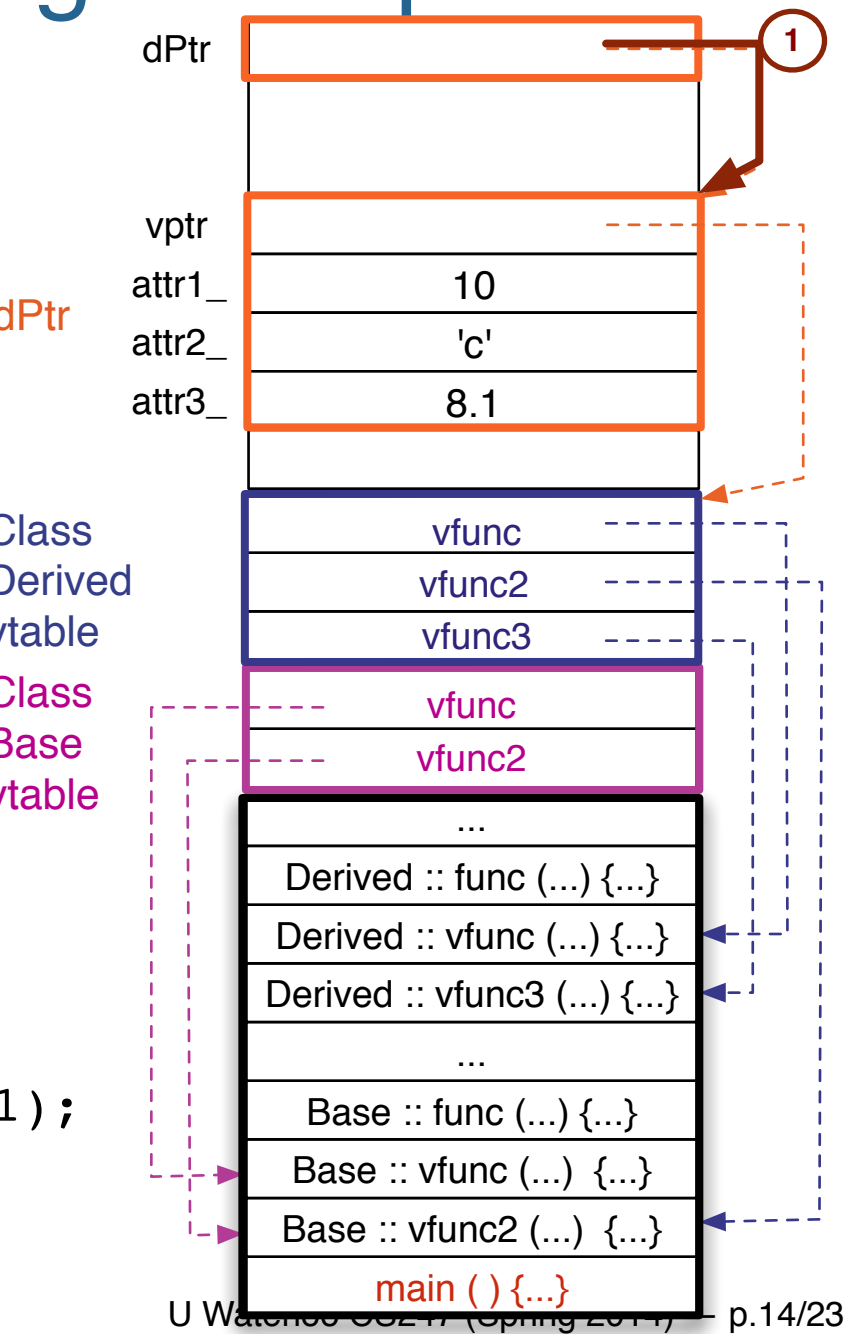


Dynamic Binding Example

Step 1: Dereference the pointer variable dPtr.

```
class Derived : public Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc3();
private:
   float attr3_;
};

int main (void) {
   Base *dPtr;
   dPtr = new Derived(10, 'c', 8.1);
   dPtr->vfunc2();
}
```

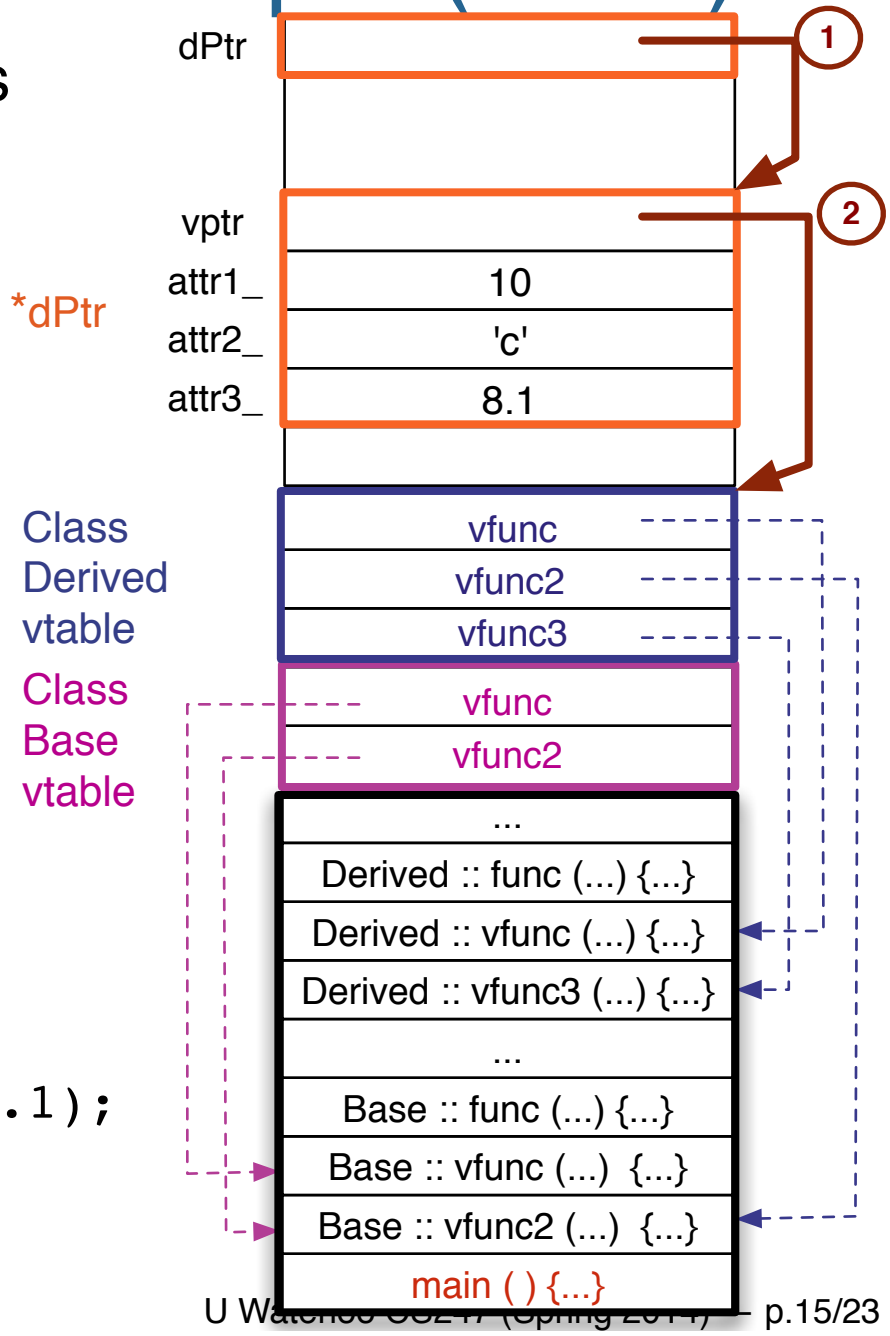


Dynamic Binding Example (cont.)

Step 2: Dereference the object's vptr pointer to access the class vtable

```
class Derived : public Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc3();
private:
   float attr3_;
};

int main (void) {
   Base *dPtr;
   dPtr = new Derived(10, 'c', 8.1);
   dPtr->vfunc2();
}
```

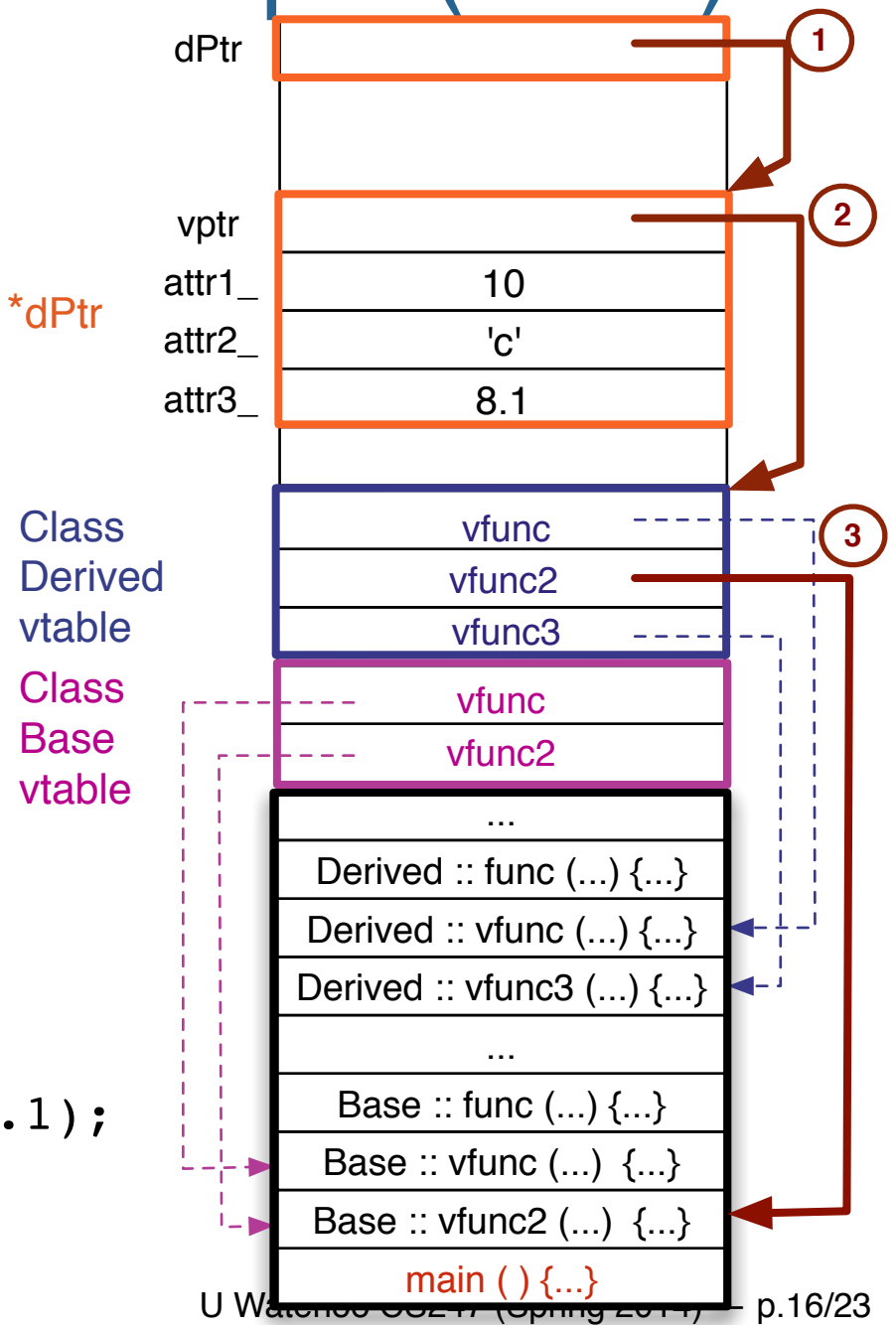


Dynamic Binding Example (cont.)

Step 3: Index into the class vtable and dereference the appropriate method pointer.

```
class Derived : public Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc3();
private:
   float attr3_;
};

int main (void) {
   Base *dPtr;
   dPtr = new Derived(10, 'c', 8.1);
   dPtr->vfunc2();
}
```

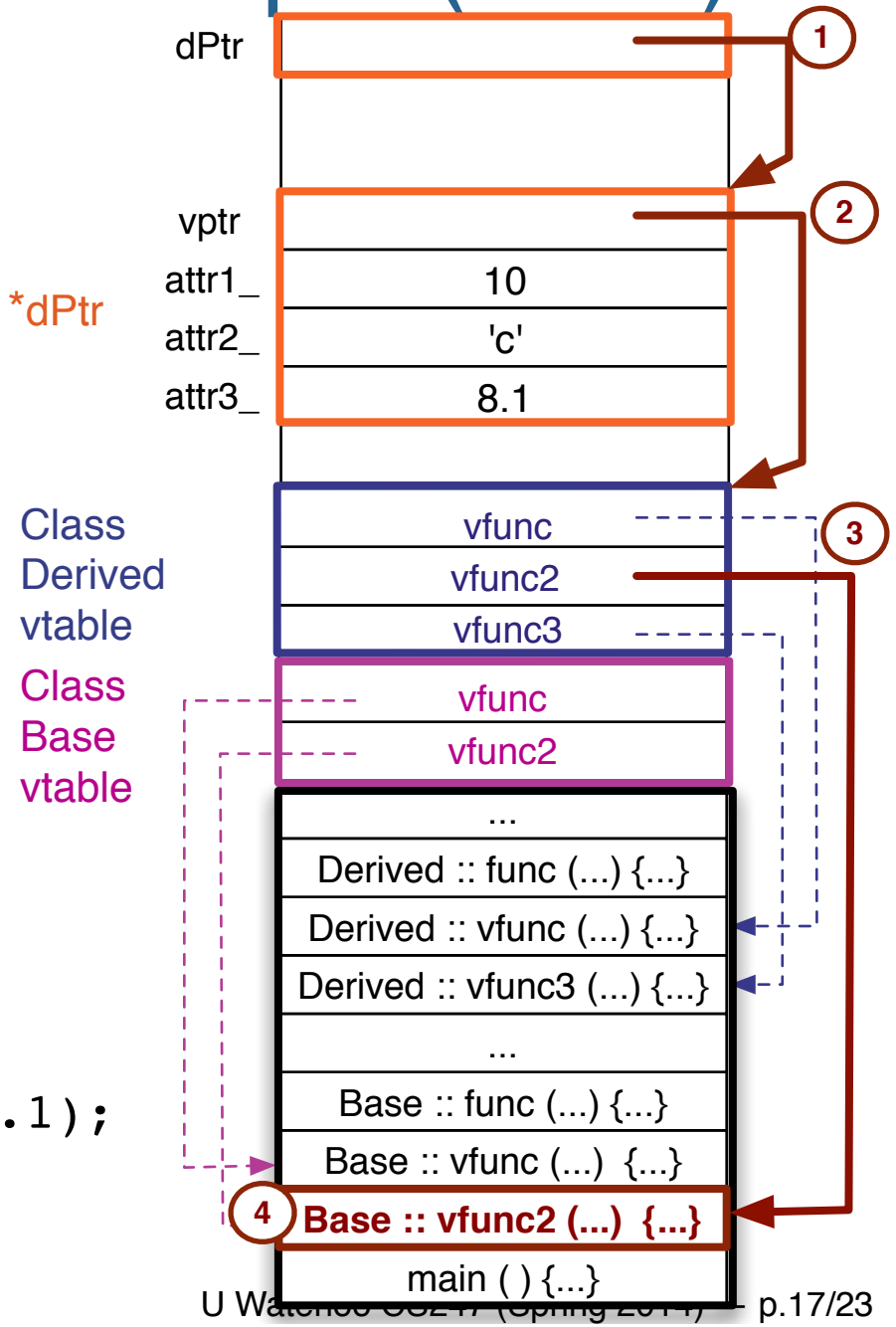


Dynamic Binding Example (cont.)

Step 4: Execute dereferenced method.

```
class Derived : public Base {
public:
   ...
   void func();
   virtual void vfunc();
   virtual void vfunc3();
private:
   float attr3_;
};

int main (void) {
   Base *dPtr;
   dPtr = new Derived(10, 'c', 8.1);
   dPtr->vfunc2();
}
```

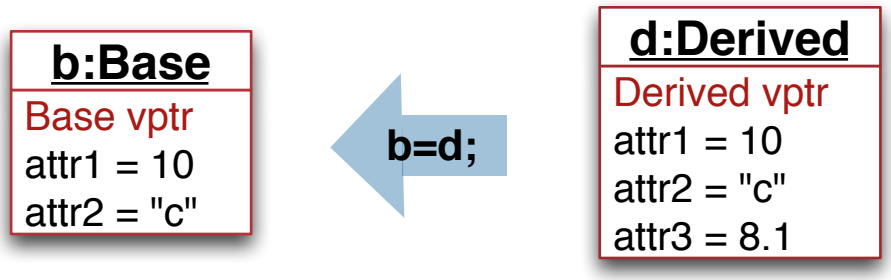


Object Slicing

```
int main (void)
{
   Base b(42, 'x');
   Derived d(10, 'c', 8.1);
   b = d; // compiles OK
   ...
}
```

Assignment operator is the base class's `operator=`

- doesn't copy the derived class's extra data members
- doesn't copy the derived class's vptr
- called object slicing



Which Methods should be Virtual?

General Convention: the decision as to whether a public method is virtual or not is typically made for the class as a whole (polymorphic class or not?), and not for individual methods.

Virtual Destructors

```
DerivedClass *d;
d = new DerivedClass();
delete d;

BaseClass *b;
b = new DerivedClass();
delete b;
```

Best Practice: If there exists any virtual method, then the class is polymorphic and should have a virtual destructor.

If we do nothing, then the compiler will generate a default non-virtual destructor for us.

Virtual print() Member Function

To print a polymorphic object, we define two functions:

- a virtual protected member function `print()` which does the actual printing
- a non-member inserter operator (`operator<<`) that takes an instance of our class and calls its member function `print()`.

Virtual print() Member Function

```
class Person {
private:
    friend ostream& operator<< (ostream &os, const Person &p);
    virtual void print(ostream&) const;
    ...
};

void Person::print(ostream &os) const {
    os << "Person{" << name << ... << "}";
}
```

```
ostream& operator<< (ostream &os, const Person &p) {
    p.print(os);
    return os;
}
```

this way, inserter op works for Person subtypes as well

```
class Student : public Person {
private:
    virtual void print(ostream&) const;
    ...
};

void Student::print(ostream &os) const
{   os << "Student{";
    Person::print(os);
    os << ", " << studentID << ", " << ... << "}";
}
```

Virtual-ness of Essential Operators

Below is a summary of how the special member functions should be declared in a polymorphic class.

```
class Person {
public:
   Person();
   Person(Person &source);
   virtual ~Person();
   void operator=(Person &source);
   virtual bool operator==(Person &source);
   virtual void print(ostream &os);
};
```