CSE 333 Section 7

STL and Inheritance

Logistics

Due Friday

Exercise 14a @ 11:00 am

Due Thursday (1 week!)

Homework 3 @ 9:00 pm

STL Question

- Work on worksheet page 6.

Inheritance

- **Derived** class inherits from the **base** class
  - In 333, we always use *public* inheritance
  - Inherits all *non-private* member variables
  - Inherits all *non-private* member functions
    except for ctor, cctor, dtor, op=
- Access specifiers revisited:
  - **Private** members cannot be accessed by derived classes
  - **Protected** members are available to base & derived

Example

```
class Base {
public:
  int x;
  Base(int x) :x(x) {}
};

class Derived : Base {
public:
  void PrintX() { std::cout << x << std::endl; }
};
```

Example

```
class Base {
public:
  int x;
  Base(int x) :x(x) {}
};

class Derived : public Base {
public:
  void PrintX() { std::cout << x << std::endl; }
};
```

Example

```
class Base {
public:
  int x;
  Base(int x) :x(x) {}
};

class Derived : public Base {
public:
  Derived(int x) :Base(x) {}
  void PrintX() { std::cout << x << std::endl; }
};
```

Example

```
class Base {
public:
  int x;
  Base(int x) :x(x) {}
};

class Derived : public Base {
public:
  using Base::Base;
  void PrintX() { std::cout << x << std::endl; }
};
```

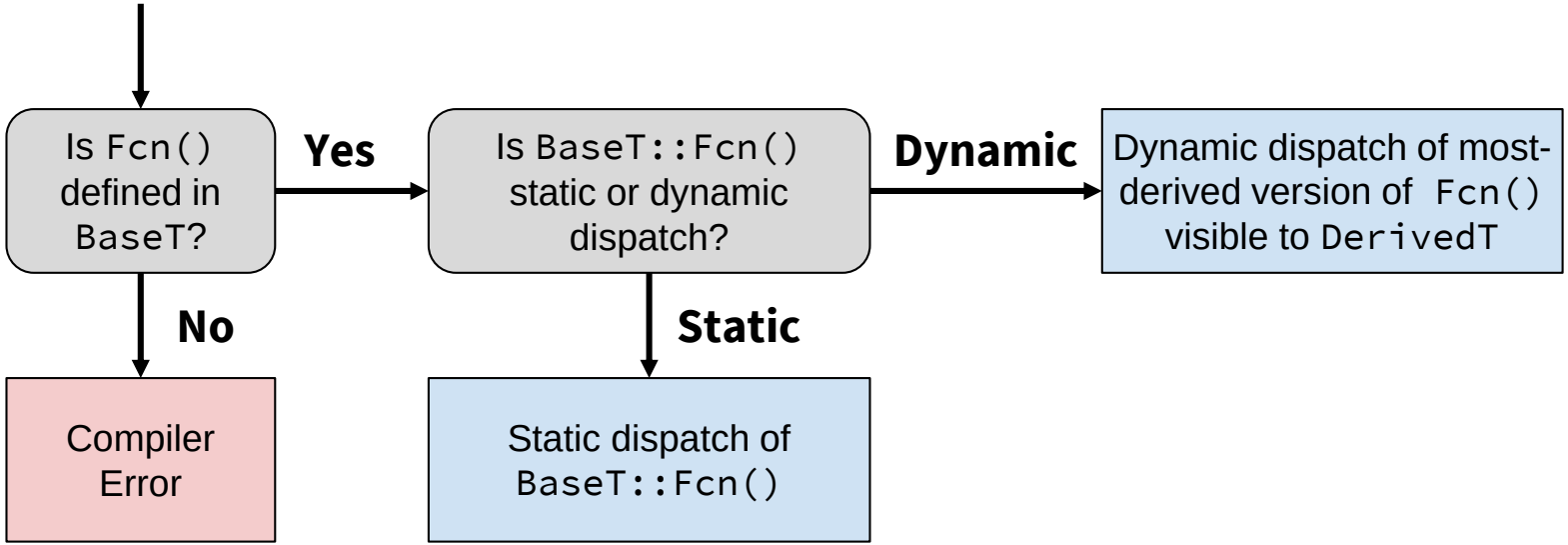
Static vs. Dynamic Dispatch

How to resolve invoking a method via a polymorphic pointer:

1. Static dispatch
    - We determine *at compile time* which implementation to call
    - Compiler generates a hard-coded call to function
2. Dynamic dispatch
    - Which implementation is determined *at runtime* via lookup
    - Compiler generates code that accesses function pointers added to the class

Dispatch Decision Tree

```
BaseT *ptr = new DerivedT();
ptr->Fcn();  // which version is called?
```



Dispatch Keywords

- `virtual` – request dynamic dispatch
  - Is “sticky”: overridden virtual method in derived class is still virtual with or without the keyword
- `override` – ensures that the function is virtual and is overriding a virtual function from a base class
  - Generates a compiler error if conditions are not met

Practice: static, dynamic, or error?

```
class Base {
  void Foo();             //
  void Bar();             //
  virtual void Baz();     //
};

class Derived : public Base {
  virtual void Foo();     //
  void Bar() override;    //
  void Baz();             //
};
```

Practice: static, dynamic, or error?

```
class Base {
  void Foo();             // static dispatch
  void Bar();             // static dispatch
  virtual void Baz();     // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();     //
  void Bar() override;    //
  void Baz();             //
};
```

Practice: static, dynamic, or error?

```
class Base {
  void Foo();             // static dispatch
  void Bar();             // static dispatch
  virtual void Baz();     // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();     // now dynamic (for more derived)
  void Bar() override;    //
  void Baz();             //
};
```

Practice: static, dynamic, or error?

```
class Base {
  void Foo();             // static dispatch
  void Bar();             // static dispatch
  virtual void Baz();     // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();     // now dynamic (for more derived)
  //void Bar() override;// compiler error
  void Bar();             // static dispatch
  void Baz();             //
};
```

Practice: static, dynamic, or error?

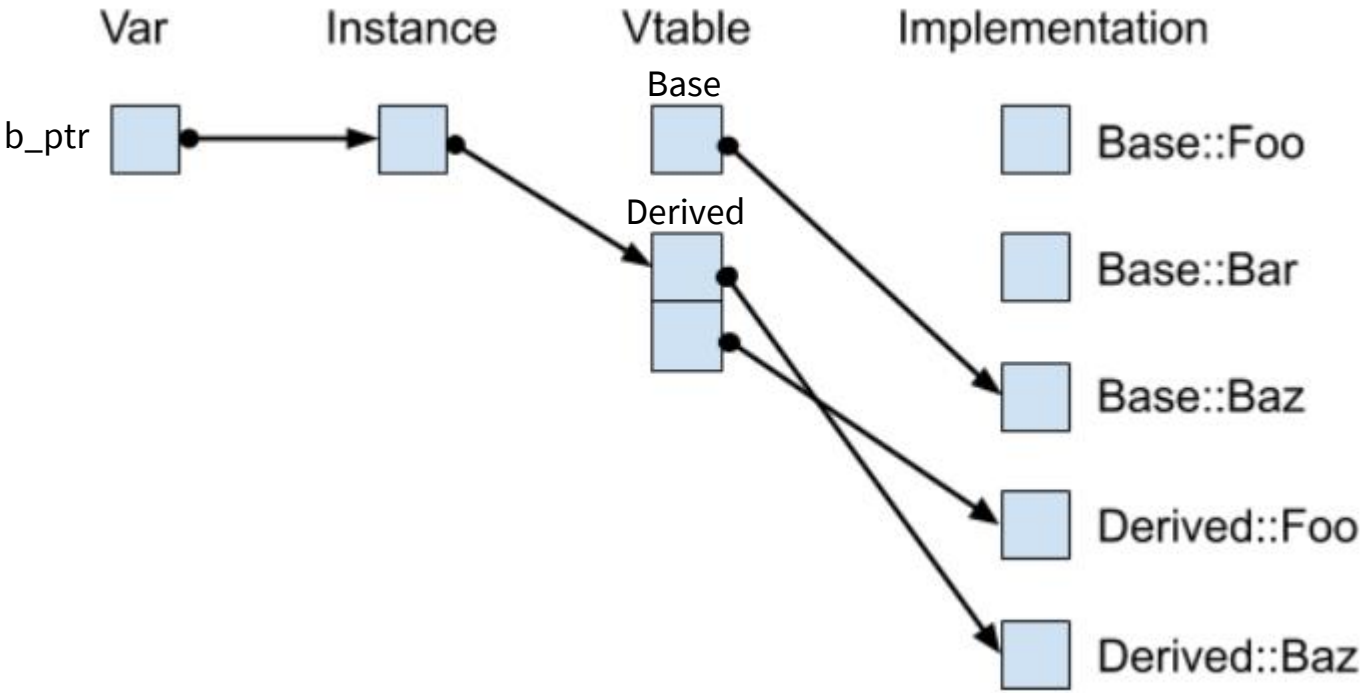
```
class Base {
  void Foo();             // static dispatch
  void Bar();             // static dispatch
  virtual void Baz();     // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();     // now dynamic (for more derived)
  //void Bar() override;// compiler error
  void Bar();             // static dispatch
  void Baz();             // still dynamic (sticky!)
};
```

Vtable (Virtual Function Table)

- An array of function pointers defined for each class that has at least one virtual method
  - Needed to enable dynamic dispatch
  - Each class has one vtable
    - Derived classes maintain ordering of vtable entries
- Each class object instance has a pointer to that vtable (vptr)
- When calling a virtual method, look it up in the vtable to see which implementation to call

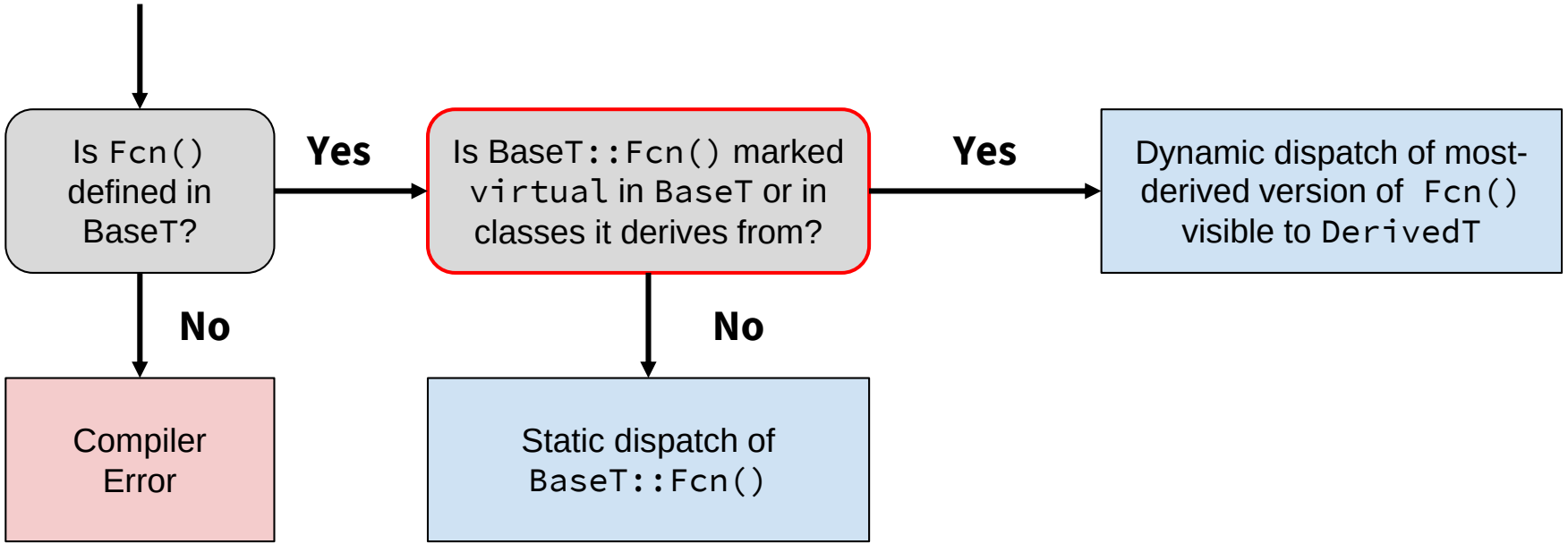
Vtable Diagrams

```
Base *b_ptr = new Derived;
```



Updated Dispatch Decision Tree

```
BaseT *ptr = new DerivedT();
ptr->Fcn();  // which version is called?
```



Dispatching Destruction

```
class Base {

};

class Derived : public Base {
  std::string x;
public:
  Derived() :x("this string is stored on the heap!") {}
  ~Derived() { std::cout << "dtor of Derived" << std::endl; }
};

Base *x = new Derived();
delete x;
```

Dispatching Destruction

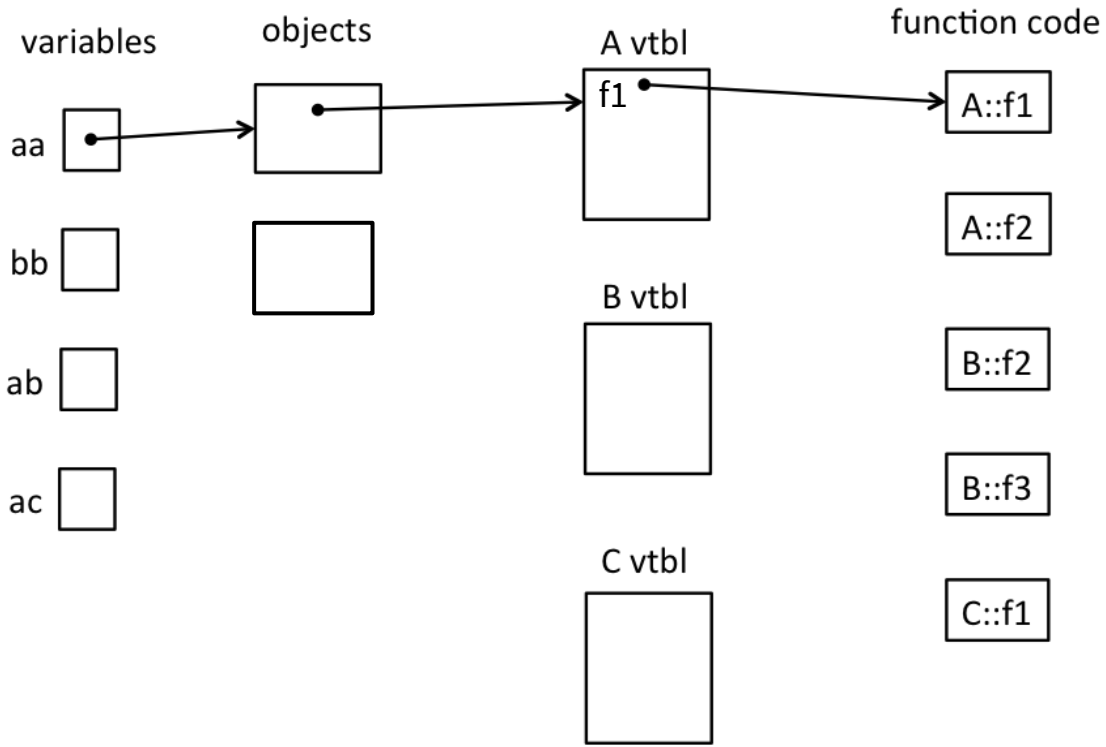
```
class Base {
  virtual ~Base() {}
};

class Derived : public Base {
  std::string x;
public:
  Derived() :x("this string is stored on the heap!") {}
  ~Derived() { std::cout << "dtor of Derived" << std::endl; }
};

Base *x = new Derived();
delete x;
```

Dispatching Destruction

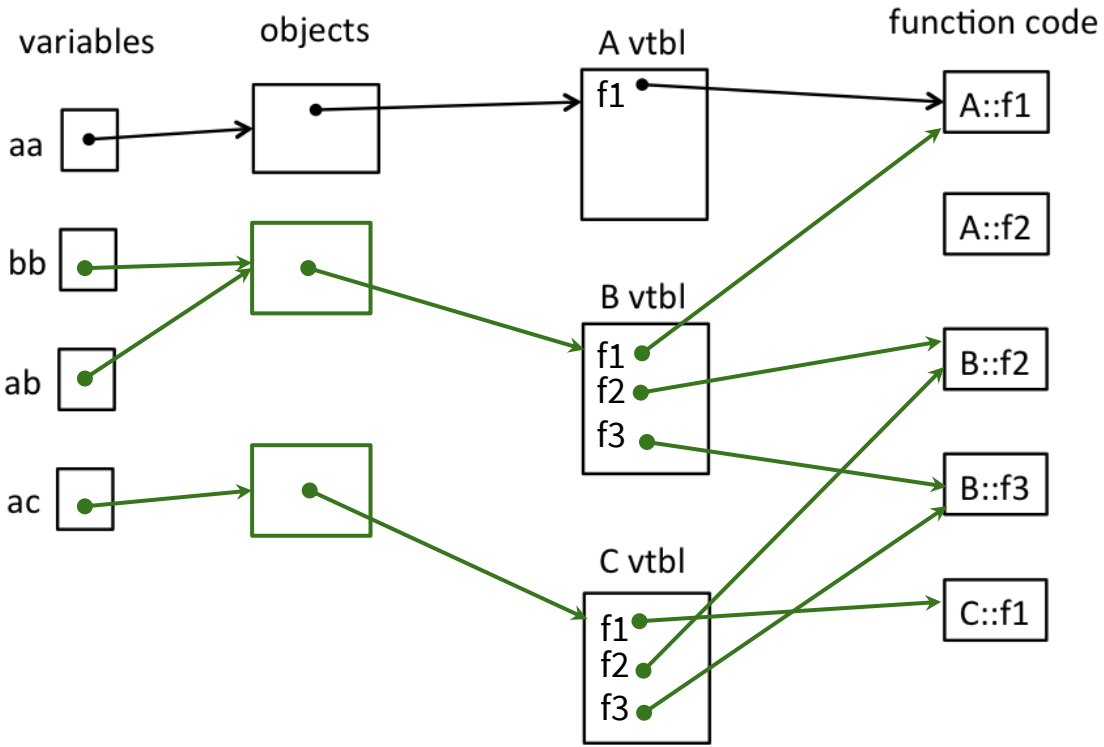
```
class Base {
  virtual ~Base() = default;
};

class Derived : public Base {
  std::string x;
public:
  Derived() :x("this string is stored on the heap!") {}
  ~Derived() { std::cout << "dtor of Derived" << std::endl; }
};

Base *x = new Derived();
delete x;
```

Exercise 1!

Exercise 1

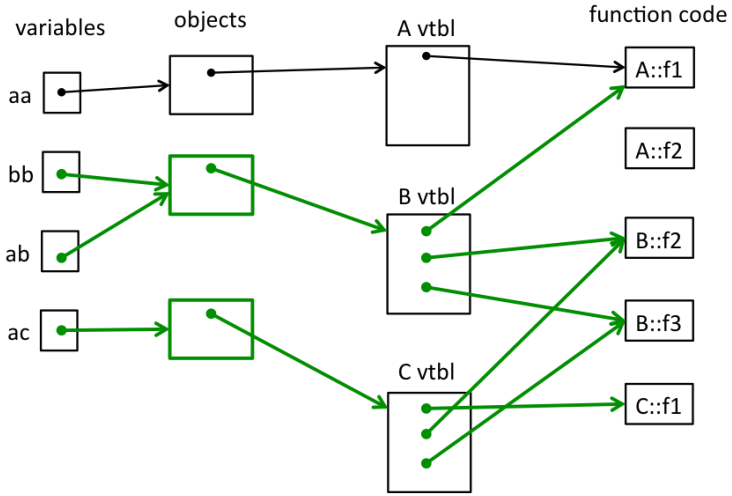

variables
objects
A vtbl
function code
aa
bb
ab
ac
f1
A::f1
A::f2
B vtbl
B::f2
B::f3
C vtbl
C::f1

Exercise 1


variables
objects
A vtbl
function code
aa
bb
ab
ac
f1
A::f1
A::f2
B vtbl
f1
f2
f3
B::f2
B::f3
C vtbl
f1
f2
f3
C::f1

Exercise 1 Solution

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

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl;
}
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl;
}
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

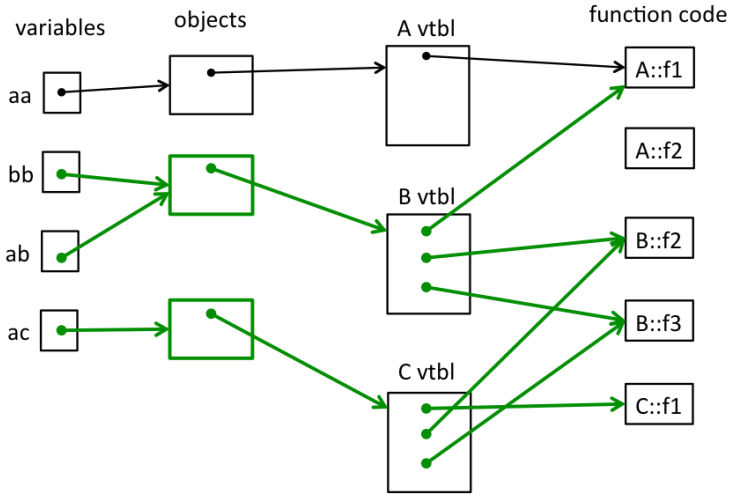



```
A* aa = new A();

aa->f1();
```

```
A::f2
A::f1
```

Exercise 1 Solution

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

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl;
}
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl;
}
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

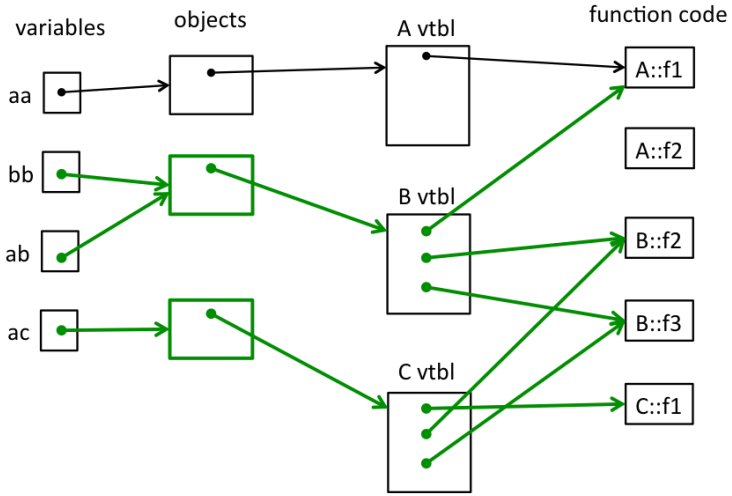



```
B* bb = new B();

bb->f1();
```

```
A::f2
A::f1
```

Exercise 1 Solution

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

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl;
}
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl;
}
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

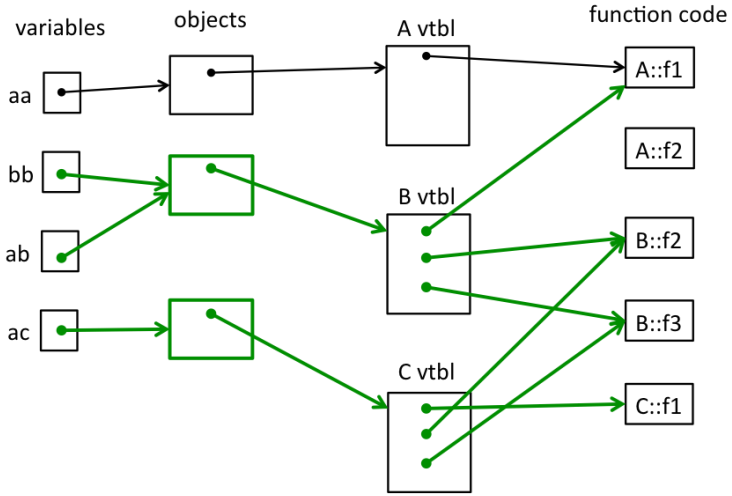



```
B* bb = new B();
A* ab = bb;

bb->f2();
cout << "----" << endl;
ab->f2();
```

```
B::f2
----
A::f2
```

Exercise 1 Solution

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

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl;
}
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl;
}
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```

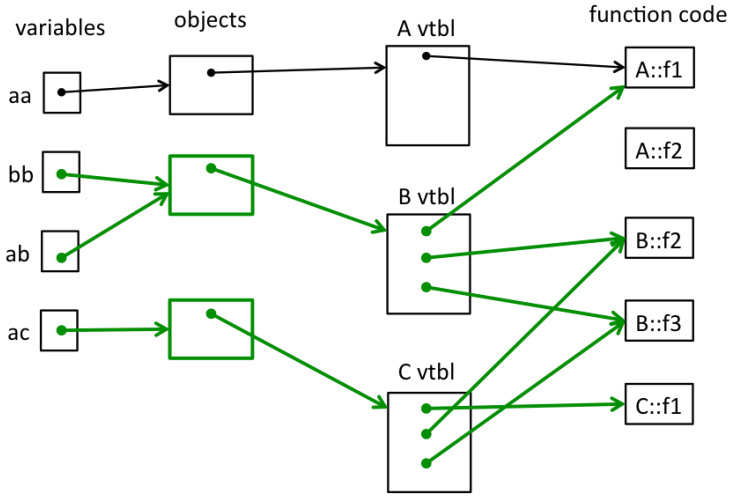



```
B* bb = new B();

bb->f3();
```

```
A::f2
A::f1
B::f3
```

Exercise 1 Solution

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

class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl;
}
  void f2() { cout << "A::f2" << endl; }
};

class B: public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl;
}
  virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
 public:
  void f1() { f2(); cout << "C::f1" << endl; }
};
```




```
A* ac = new C();

ac->f1();
```

```
B::f2
C::f1
```