CSE 333 Section 7

Casting and Inheritance

Casting

https://me.me/i/when-you-mistype-a-keyword-in-c-static-cat-dynamic-2bcd3ae67b1343f58403678461cec7b9

Casting in C++

Four different casts that are more explicit:

1. `static_cast<to_type>(expression)`
2. `dynamic_cast<to_type>(expression)`
3. `const_cast<to_type>(expression)`
4. `reinterpret_cast<to_type>(expression)`

When programming in C++, you should use these casts!

Static Cast

```
static_cast<to_type>(expression)
```

Used to:

1) Convert pointers of *related* types

```
Base* b = static_cast<Base*>(new Derived);
```

- compiler error if types aren't related

2) Non-pointer conversion

```
int qt = static_cast<int>(3.14);
```

Static Cast

```
static_cast<to_type>(expression)
```

[!] Be careful when *casting up*:

```
Derived* d = static_cast<Derived*>(new Base);
d->y = 5;
```

- compiler will let you do this
- dangerous if you want to do things defined in **Derived**, but not in **Base**!

Dangerous Possible Outcomes

- Dynamically-dispatched method call will read past end of vtable
- Data member access will access data outside of object instance

[!] Be careful when *casting up*:

```
Derived* d = static_cast<Derived*>(new Base);
d->y = 5; // Error if base did not define “y”
```

Dynamic Cast

Note: Only works in regards to inheritance

```
dynamic_cast<to_type>(expression)
```

Used to:

1) Convert pointers of *related* types

```
Base* b = dynamic_cast<Base*>(new Derived);
```

- *compiler* error if types aren't related
- at *runtime*, returns **nullptr** if it is actually an unsafe upwards cast:

```
Derived* d = dynamic_cast<Derived*>(new Base);
```

Const Cast

```
const_cast<to_type>(expression)
```

Used to:

1) Add or remove const-ness

```
const int x = 5;
const int* ro_ptr = &x;
int* ptr = const_cast<int*>(ro_ptr);
(*ptr)++; // compiles
```

Note: Adding const to types does not require this cast

Reinterpret Cast

```
reinterpret_cast<to_type>(expression)
```

Used to:

1) Cast between *incompatible* types

```
int* ptr = 0xDEADBEEF;
int64_t x = reinterpret_cast<int64_t>(ptr);
```

- types must be of same size
- refuses to do float-integer conversions

More details:

https://google.github.io/styleguide/cppguide.html#Casting

Exercise 1

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

```
class Derived : public Base {
 public:
  int y;
}
```

```
int64_t x = 0x7ffffffffe870;
char* str = ____________________________ (x);
```

```
void foo(Base* b) {
  Derived* d = ____________________________ (b);
  // additional code omitted
}
```

```
Derived* d = new Derived;
Base* b = ____________________________ (d);
```

```
double x = 64.382;
int64_t y = ____________________________ (x);
```

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

```
class Derived : public Base {
 public:
  int y;
}
```

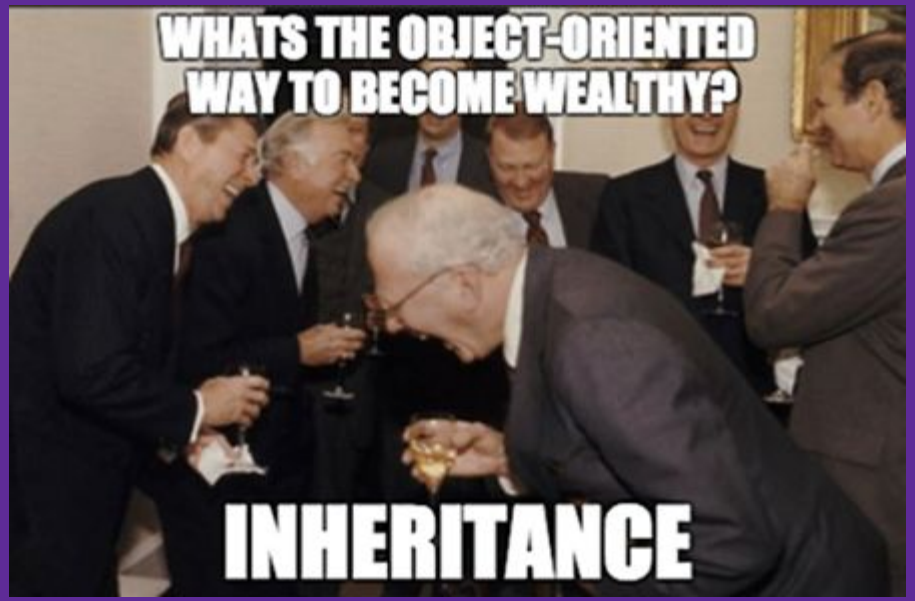
```
int64_t x = 0x7ffffffffe870;
char* str = reinterpret_cast<char*>(x);
```

```
void foo(Base* b) {
  Derived* d = dynamic_cast<Derived*>(b);
  // additional code omitted
}
```

```
Derived* d = new Derived;
Base* b = static_cast<Base*>(d);
```

```
double x = 64.382;
int64_t y = static_cast<int64_t>(x);
```

Inheritance



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

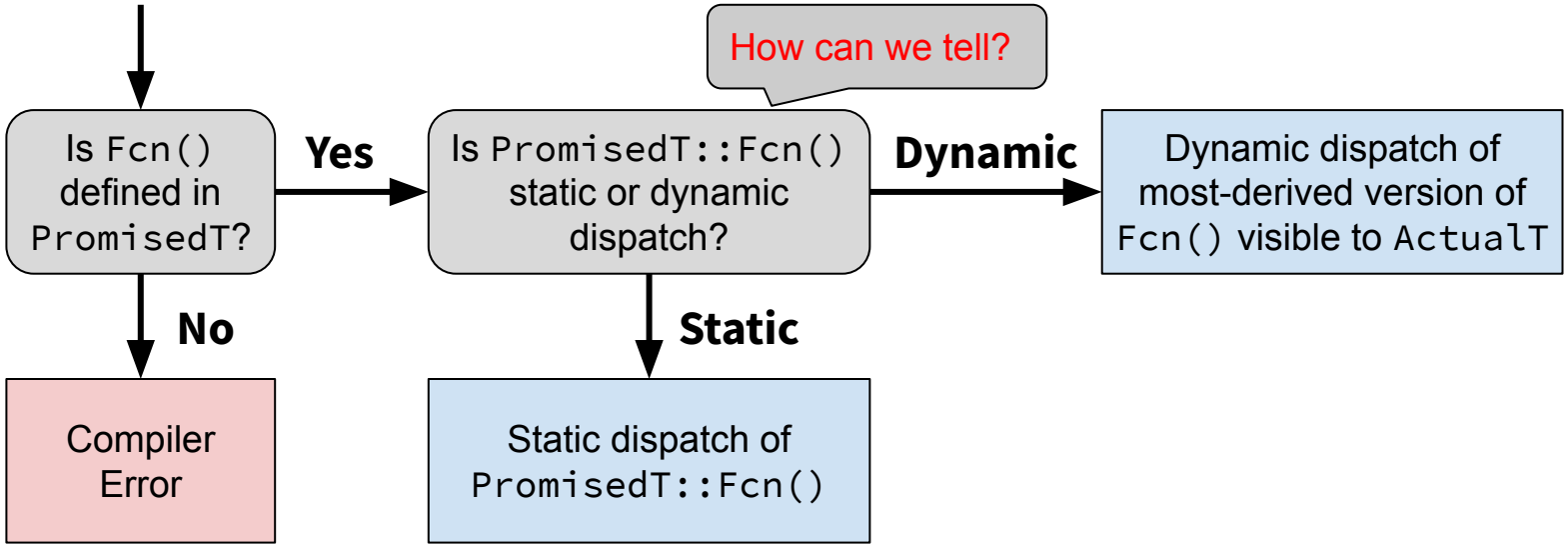
Static vs. Dynamic Dispatch

How to resolve invoking a method via a polymorphic pointer:

1. Static dispatch
    - Default behavior in C++
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

```
PromisedT* ptr = new ActualT();
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 (`@override` in Java)
  - Generates a compiler error if conditions are not met
  - Catches overloading vs. overriding bugs at compile time

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();              //
  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();     //
};

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?

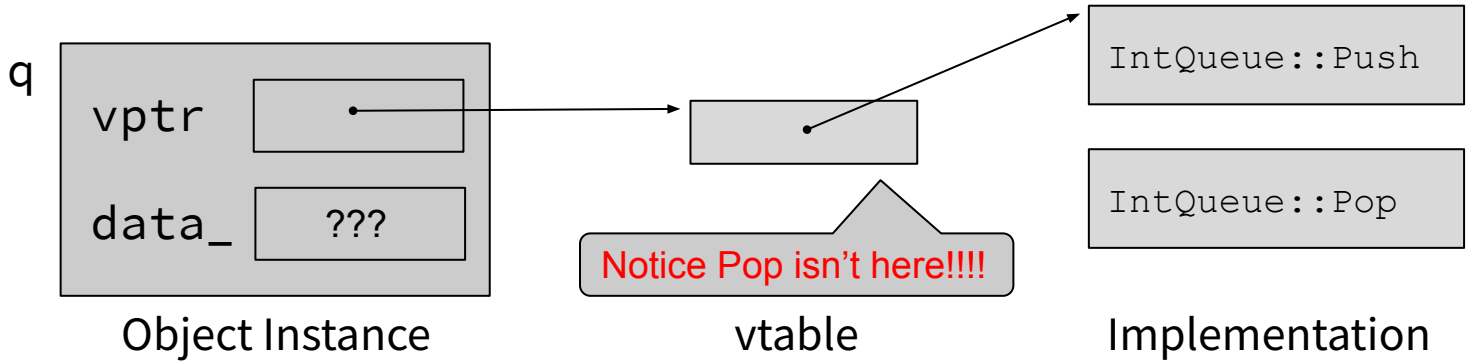
```
class Base {
  void Foo();             // static dispatch
  void Bar();             // static dispatch
  virtual void Baz();     // dynamic dispatch
};

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) & Vptr (Vtable pointer)

- vtable: An array of function pointers defined for each class that has at least one virtual method to enable dynamic dispatch
  - One per class
- vptr: Each class object instance has a pointer to that vtable
  - One per object instance

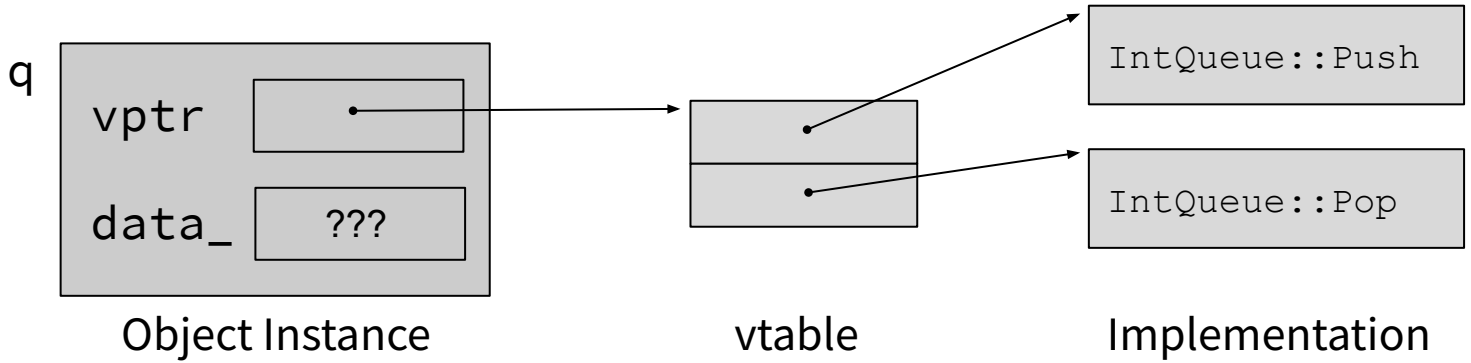
Vtable Diagrams

```
class IntQueue {
 public:
  virtual void Push(int x);
           int Pop();
 private:
  vector<int> data_;
};
IntQueue q;
```



Vtable Diagrams

```
class IntQueue {
 public:
  virtual void Push(int x);
  virtual int Pop();
 private:
  vector<int> data_;
};
IntQueue q;
```

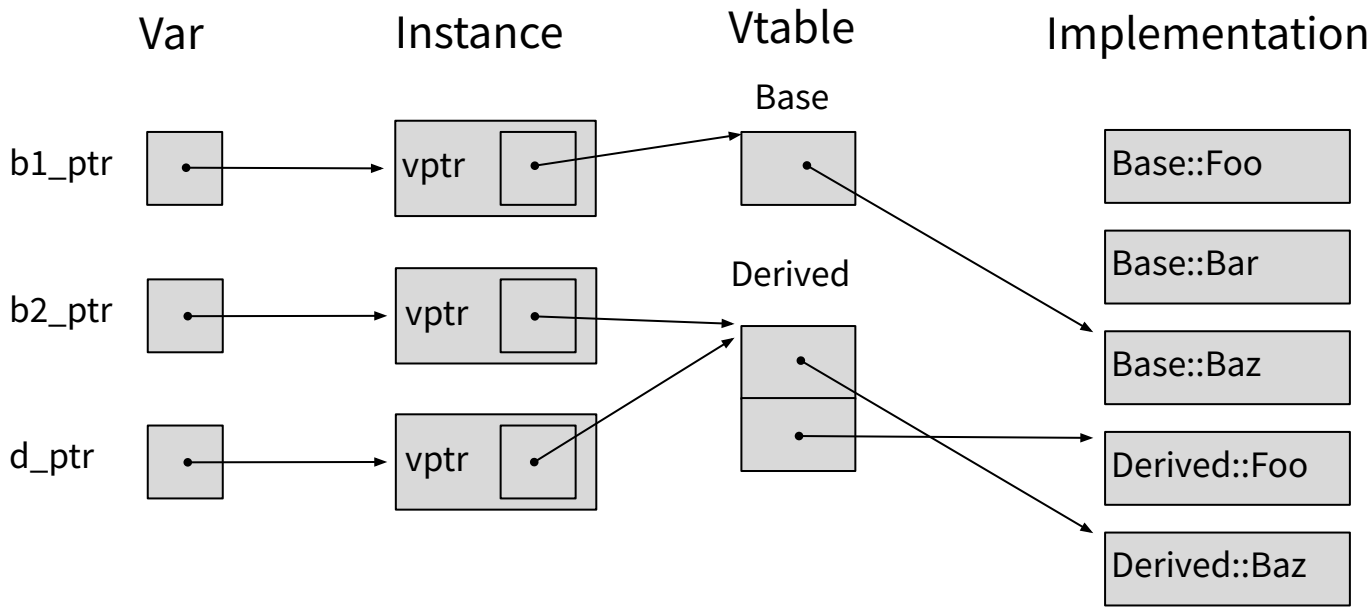


Vtable Diagrams

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

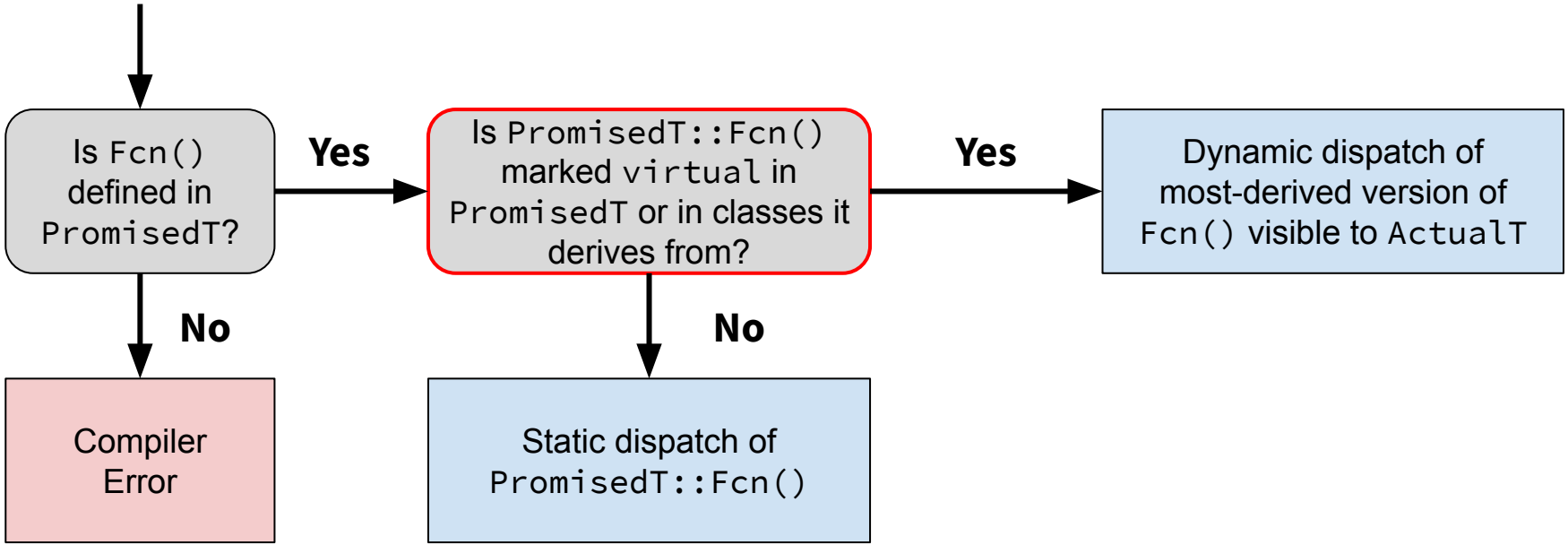
```
class Derived :
public Base {
  virtual void Foo();
  void Baz();
};
```

```
Base *b1_ptr = new Base;
Base *b2_ptr = new Derived;
Derived *d_ptr = new Derived;
```



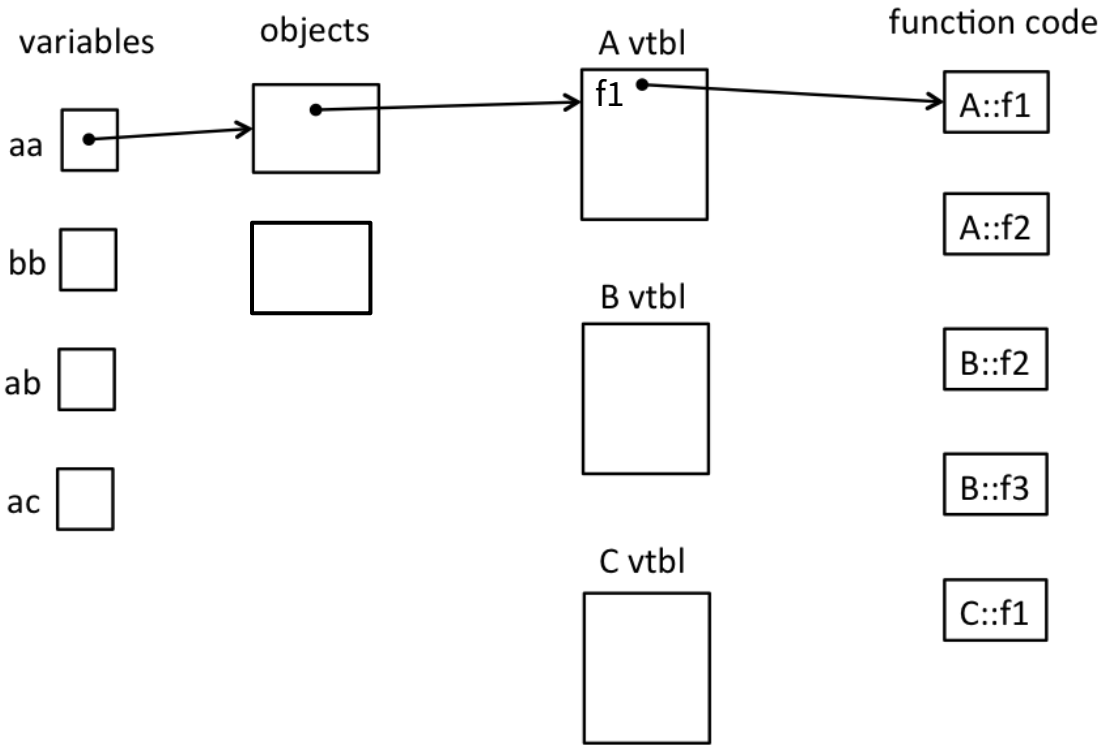
Updated Dispatch Decision Tree

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



Exercise 2

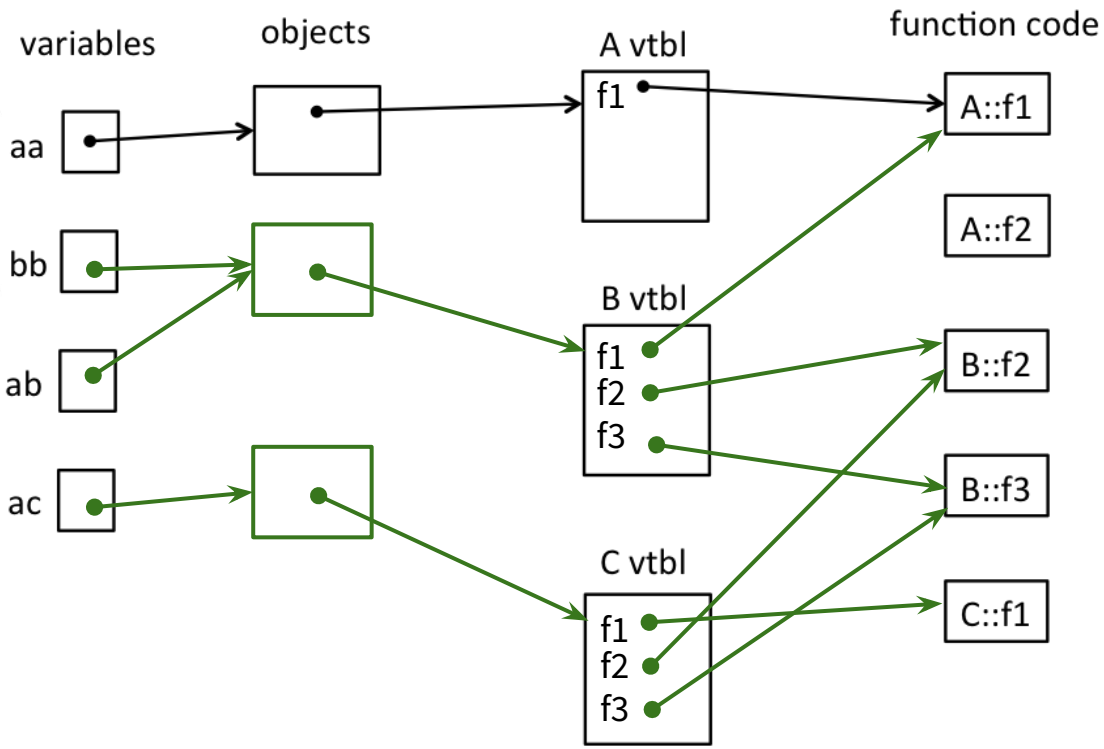
Exercise 2


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

Exercise 2 Solution (pointers)

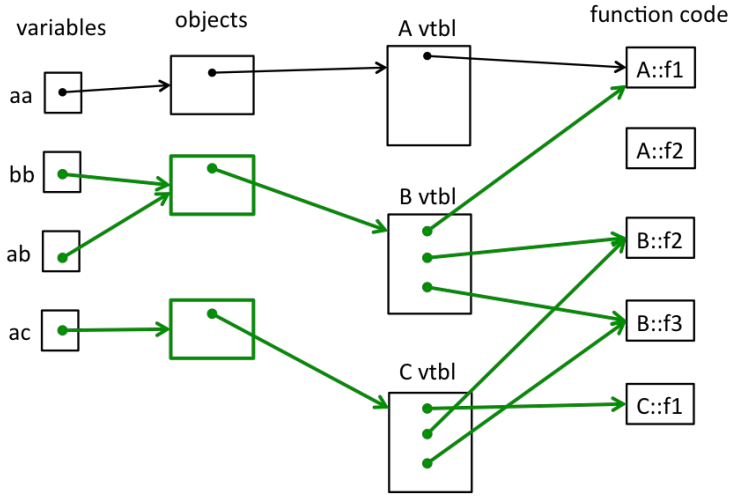
```
class A {
 public:
  virtual void f1() { f2(); cout << "A::f1" << endl; }
  void f2() { cout << "A::f2" << endl; }
};
class B : public A {
 public:
  virtual void f3() { f1(); cout << "B::f3" << endl; }
  virtual void f2() { cout << "B::f2" << endl; }
};
class C : public B {
 public:
  void f1()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

```
int main() {
  A* aa = new A();
  B* bb = new B();
  A* ab = bb;
  A* ac = new C();
```



Exercise 2 Solution (output)

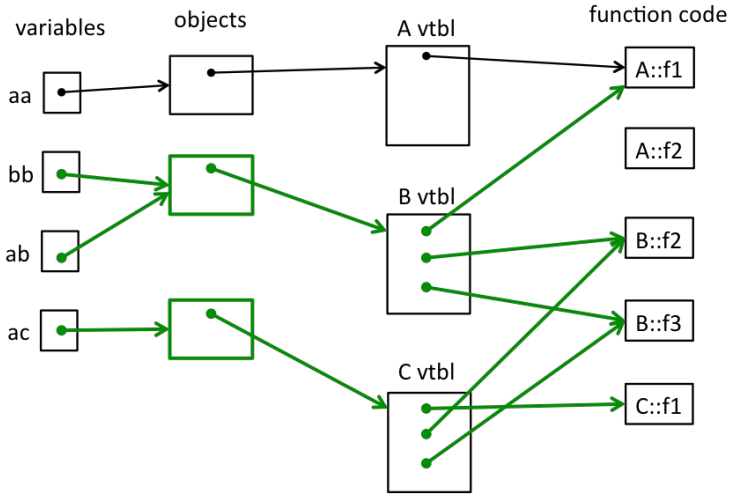
```
#include <iostream>
using namespace std;

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();
```

Exercise 2 Solution (output)

```
#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; }
};
```

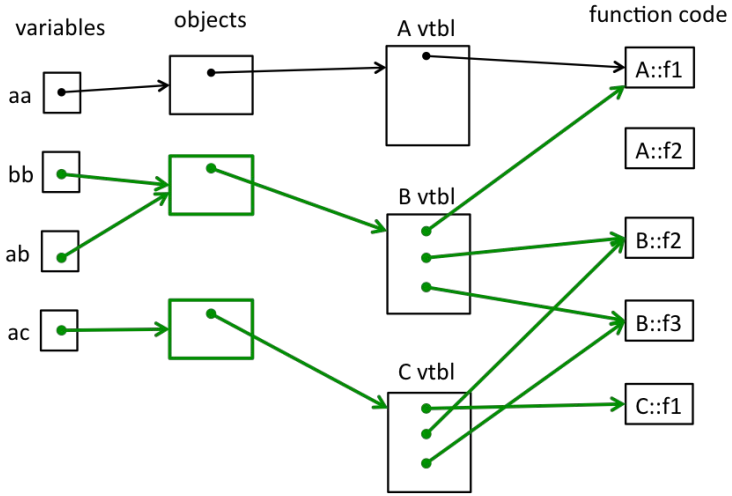



```
A* aa = new A();

aa->f1();
```

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

Exercise 2 Solution (output)

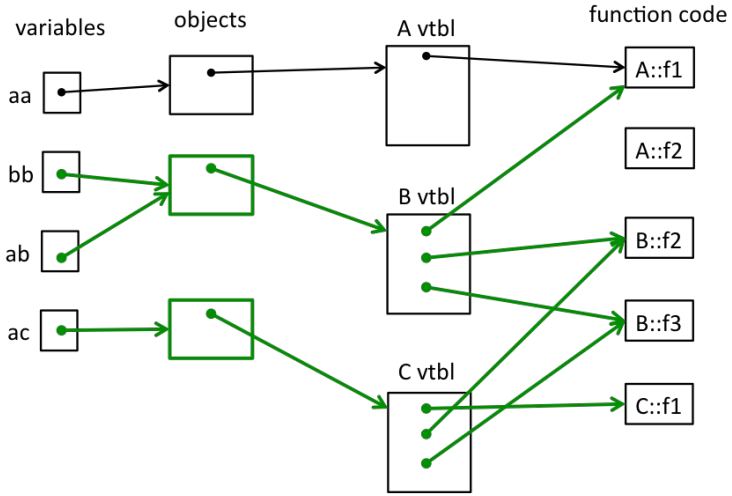
```
#include <iostream>
using namespace std;

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();
```

Exercise 2 Solution (output)

```
#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; }
};
```

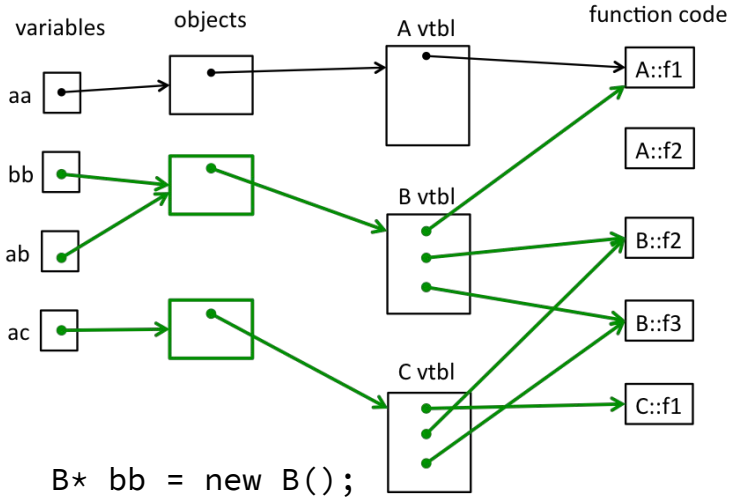



```
B* bb = new B();

bb->f1();
```

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

Exercise 2 Solution (output)

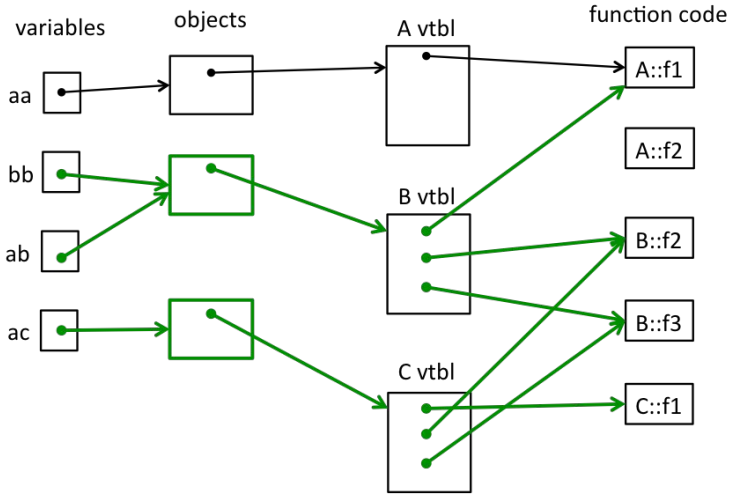
```
#include <iostream>
using namespace std;

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();
```

Exercise 2 Solution (output)

```
#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; }
};
```

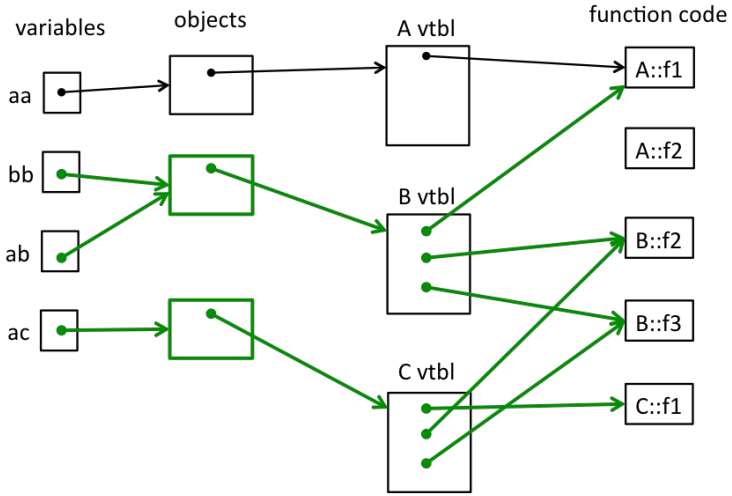



```
B* bb = new B();
A* ab = bb;

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

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

Exercise 2 Solution (output)

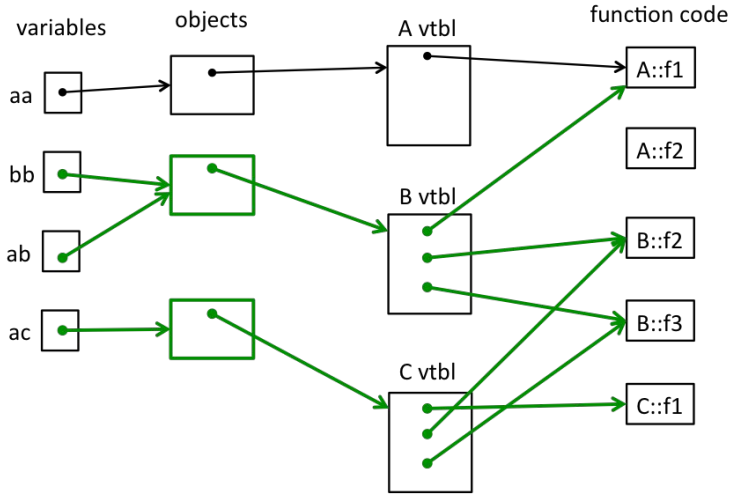
```
#include <iostream>
using namespace std;

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();
```

Exercise 2 Solution (output)

```
#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; }
};
```

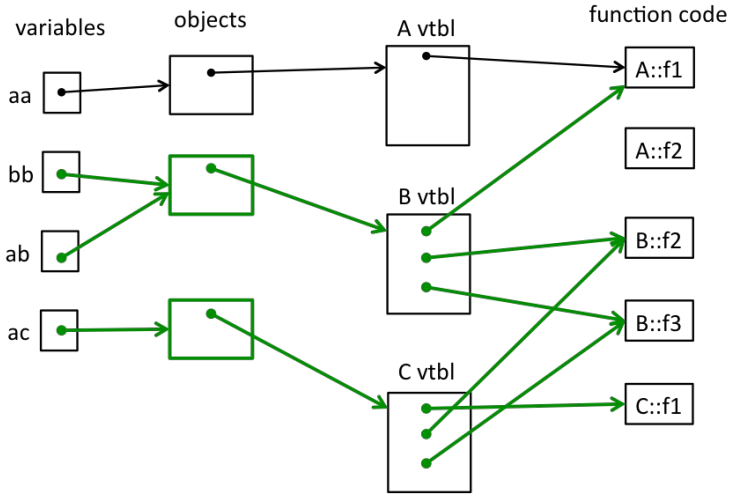



```
B* bb = new B();

bb->f3();
```

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

Exercise 2 Solution (output)

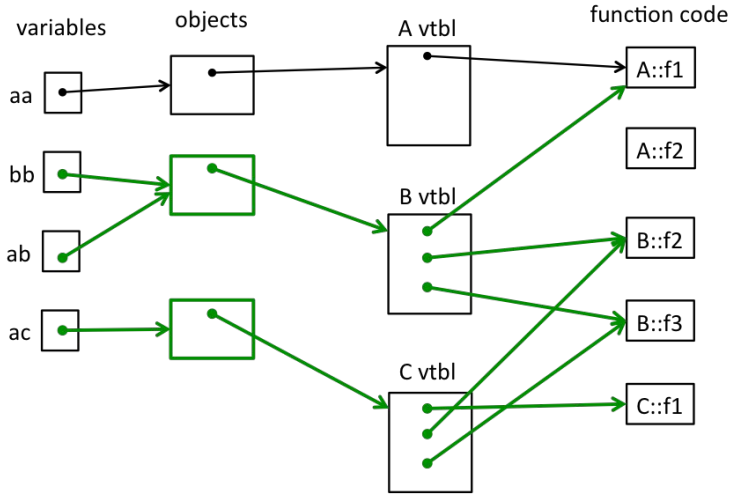
```
#include <iostream>
using namespace std;

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();
```

Exercise 2 Solution (output)

```
#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
```

Thanks for virtually attending section!