

CSE 333

Section 6

Casting and Inheritance

Logistics

- Exercise 9 due Wednesday, August 4th
- HW3 due next Thursday, August 5th
 - *A week from today*



Casting

When you mistype a keyword in C++



Static Cat



Dynamic Cat



Const Cat



Reinterpret Cat

Ever have a moment like this
when programming?

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

Casting in C vs C++

In C, we had simple casting syntax: `lhs = (new_type) rhs;`

- Convert between pointers of arbitrary type
 - Doesn't change data, just treats it differently
- Forcibly convert a primitive type to another
 - Actually changes representation

Sometimes, the intent of C-style casting is not clear!

SO, we have fancy new C++ casting!

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 ALWAYS use these casts!

- Intent is clearer
- Easier to find in code via searching

Static Cast

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

For any WELL-DEFINED conversion. Used to:

1) Convert pointers of *related* types

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

- compiler error if types aren't related

2) Cast `void*` to `T*`

Static Cast

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

3) Non-pointer conversion

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

`Static_cast` is checked at COMPILE TIME!

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: Convert pointers of *related* types, convert references 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);
```

Dynamic Cast

```
void bar() {  
    Base b; Der1 d;
```

✓ `Base* bptr = dynamic_cast<Base*>(&d);`

✓ `Der1* dptr = dynamic_cast<Der1*>(bptr);`

```
    bptr = &b;
```

✗ `dptr = dynamic_cast<Der1*>(bptr);`
}

```
class Base {  
    public:  
        virtual void foo() { }  
        float x;  
};
```

```
class Der1 : public Base {  
    public:  
        char x;  
};
```

Casts from base to derived fail at run time if the pointed-to object is not the derived type

Const Cast

`const_cast<to_type>(expression)`

Used to: 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

Const Cast

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

Mainly useful if you're trying to use badly written code!

If you need to use a function that doesn't change a parameter, BUT doesn't mark it as const, you can remove const from a variable before using it in that function.

Reinterpret Cast

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

Used to: 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

Use any other C++ cast if you can!

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 = 0x7ffffffffffe870;  
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;  
}
```

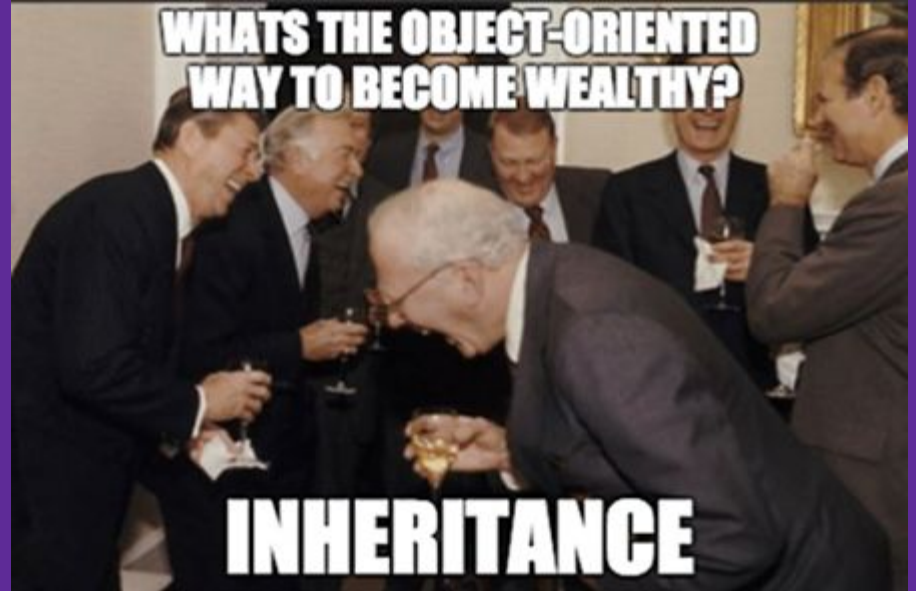
```
int64_t x = 0x7ffffffffffe870;  
char* str = reinterpret_cast<char*> (x);
```

```
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

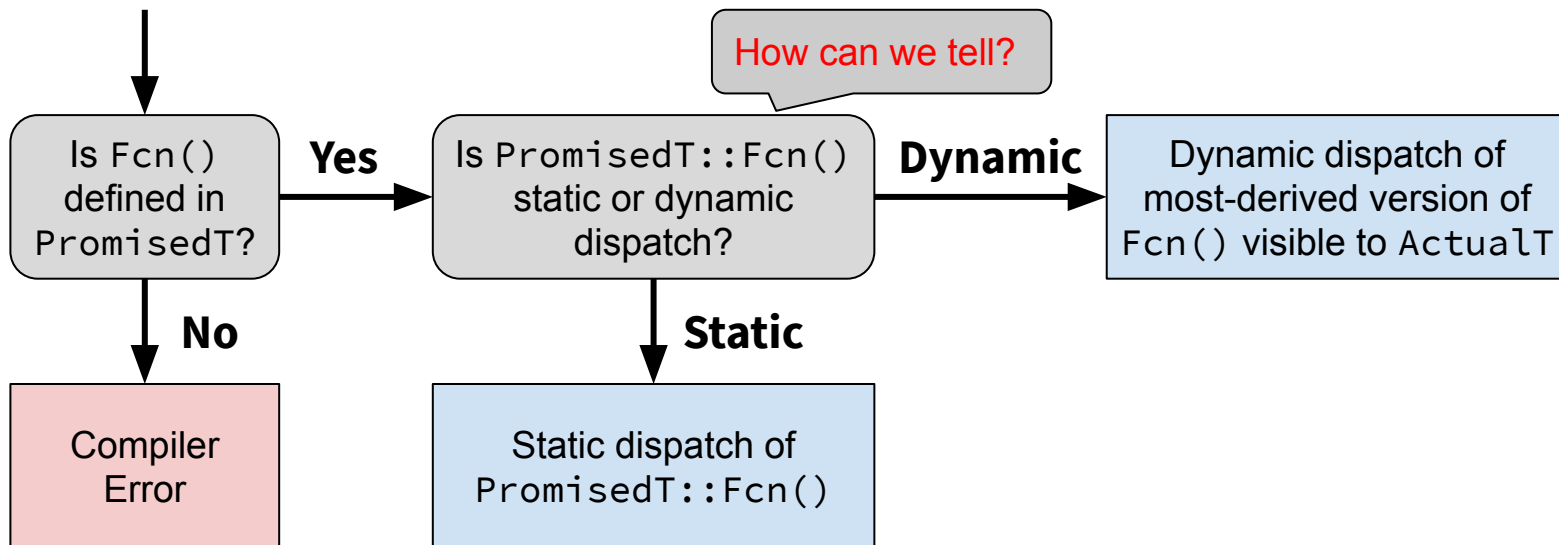
Static vs. Dynamic Dispatch

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?



yes



no



no slower

Static dispatch

Dynamic dispatch

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?



yes

Static dispatch



no

Dynamic dispatch



no slower

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?



yes

Static dispatch



no

Dynamic dispatch



no slower

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?



yes

Static dispatch



no

Dynamic dispatch



no slower

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?



yes

Static dispatch



no

Dynamic dispatch



no slower

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?



yes

Static dispatch



no

Dynamic dispatch



no slower

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?



yes

Static dispatch



no

Dynamic dispatch



no slower

Error

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

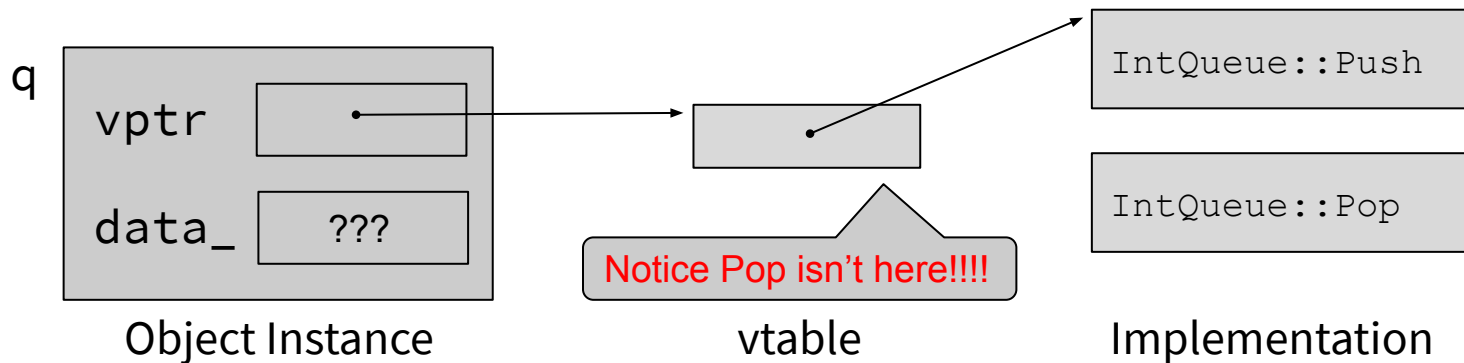
```
class Derived : public Base {  
    virtual void Foo();  // now dynamic (for more derived)  
    //void Bar() override; // compiler error  
    void Bar();         // static dispatch  
    void Baz();        // still dynamic (sticky!)  
};
```

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

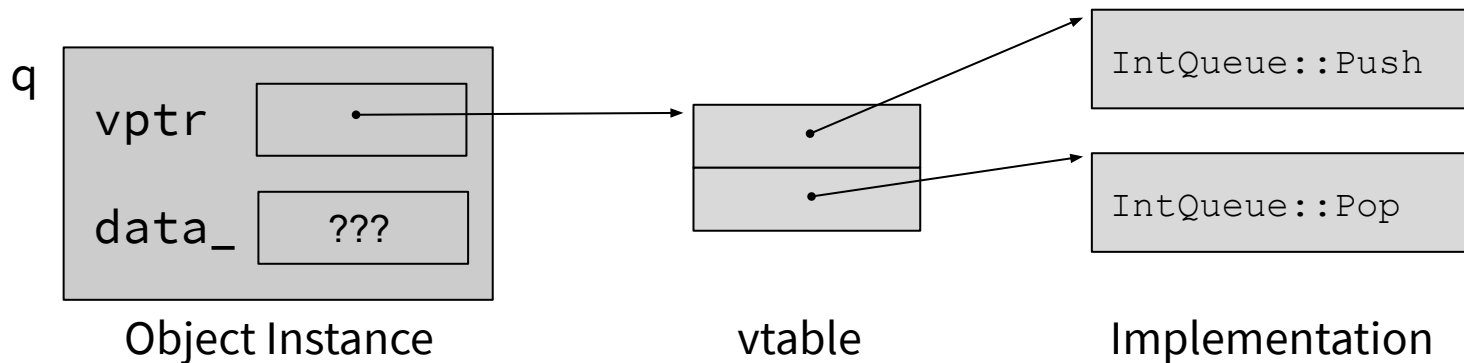
Vtable Diagrams

```
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;
```

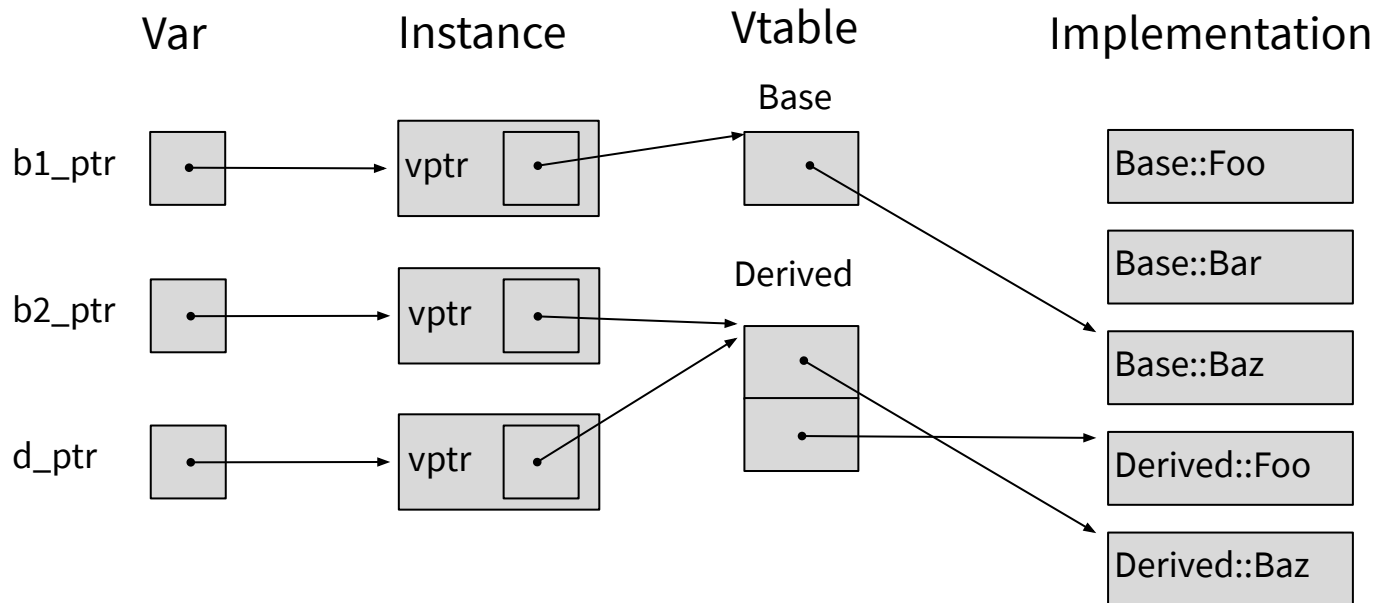


Vtable Diagrams

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

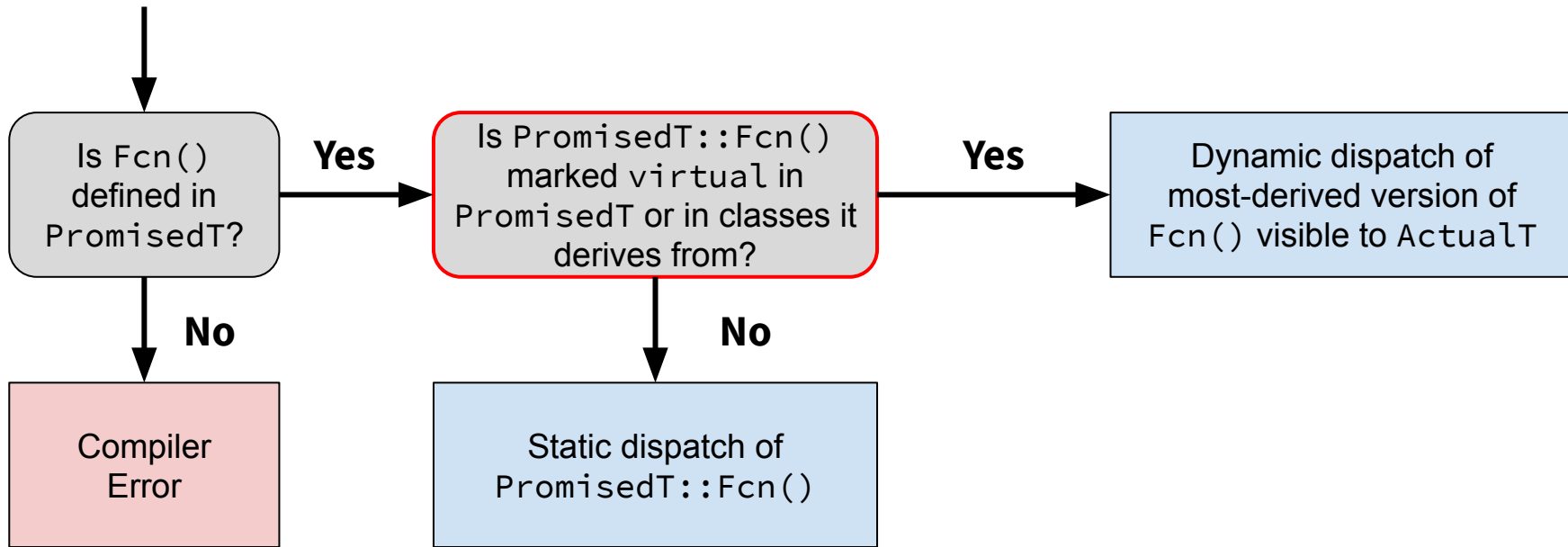
```
class Base {
    void Foo();
    void Bar();
    virtual void Baz();
};
```

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



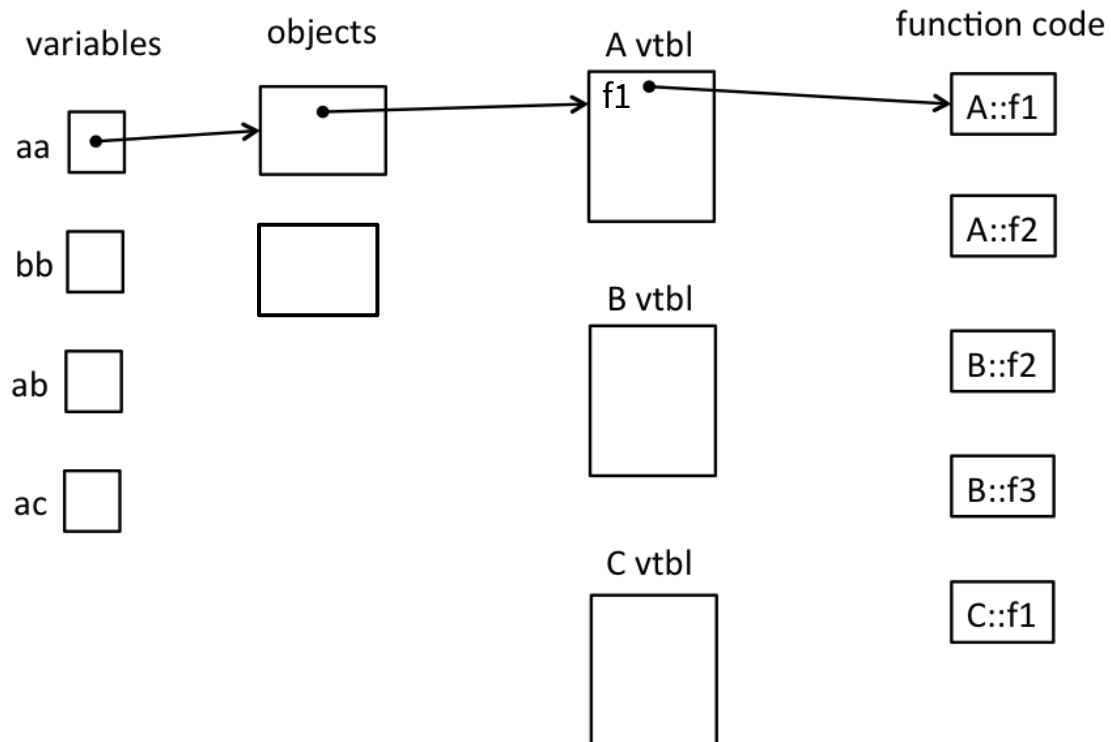
Updated Dispatch Decision Tree

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



Exercise 2

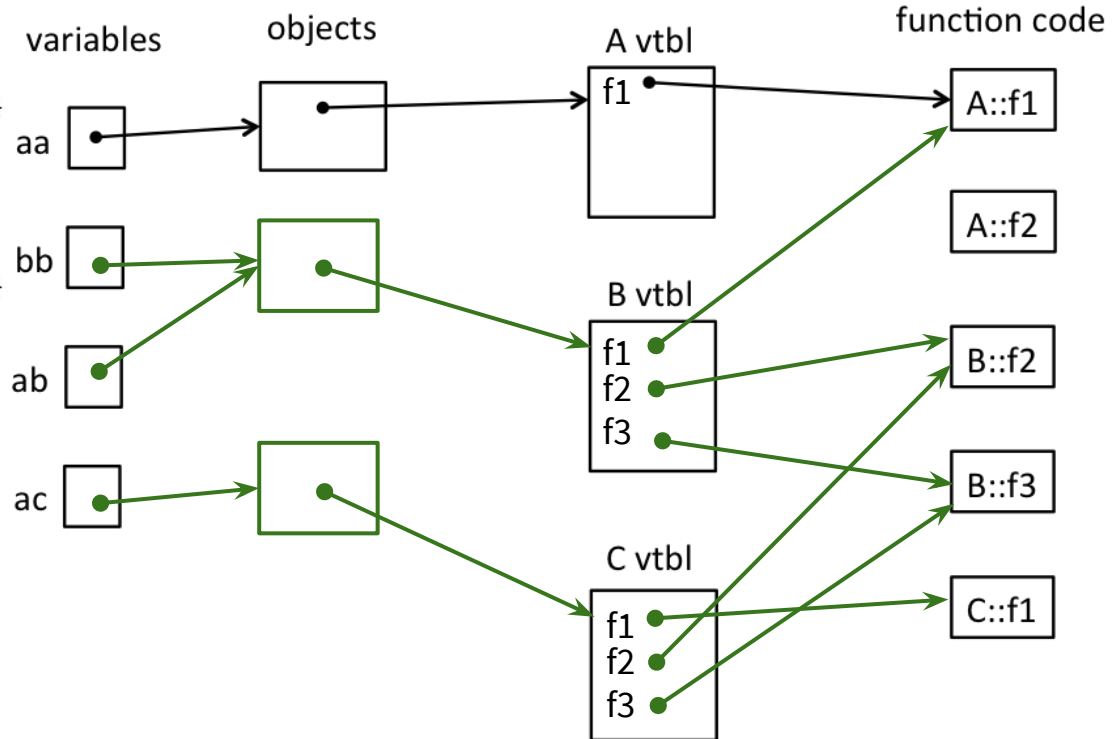
Exercise 2



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();
```



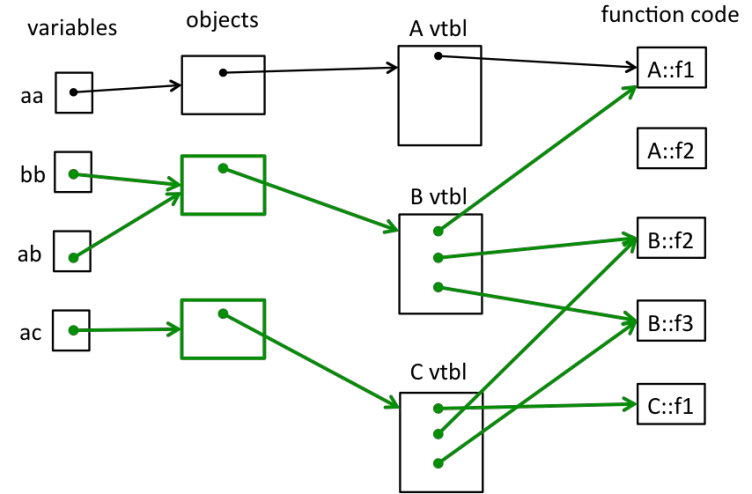
Exercise 2 Solution (output)

```
#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();
```

B::f2	A::f2	A::f2	B::f2
A::f1	C::f1	A::f1	C::f1

yes no go slower go faster

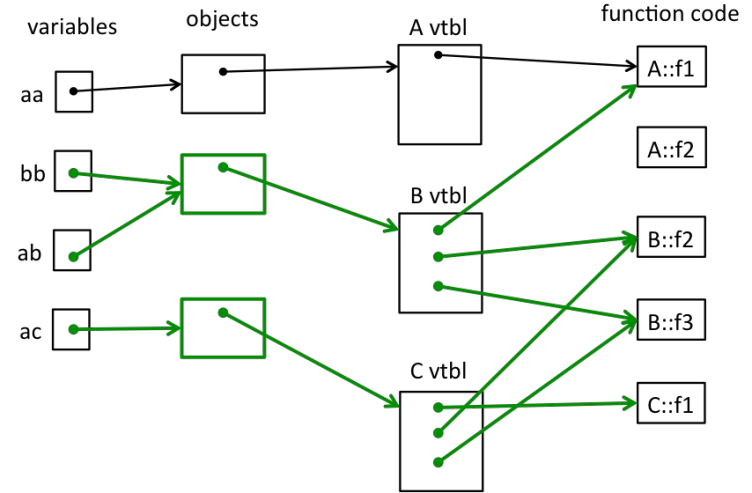
Exercise 2 Solution (output)

```
#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
```

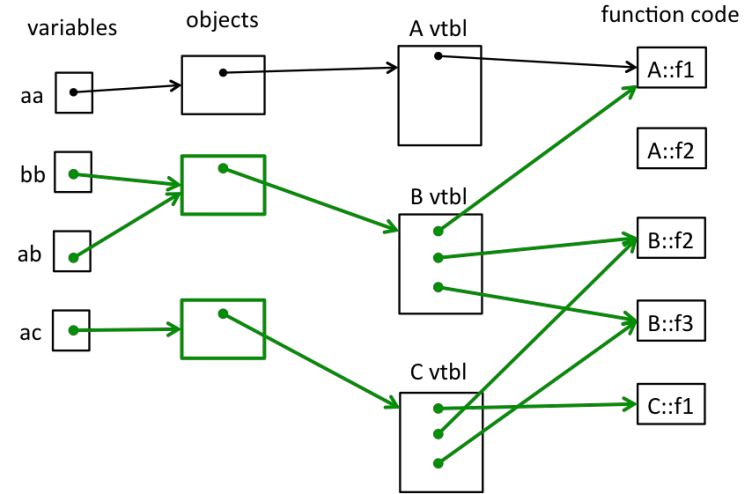
Exercise 2 Solution (output)

```
#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();
```

B::f2	A::f2	A::f2	B::f2
A::f1	C::f1	A::f1	C::f1

yes no go slower go faster

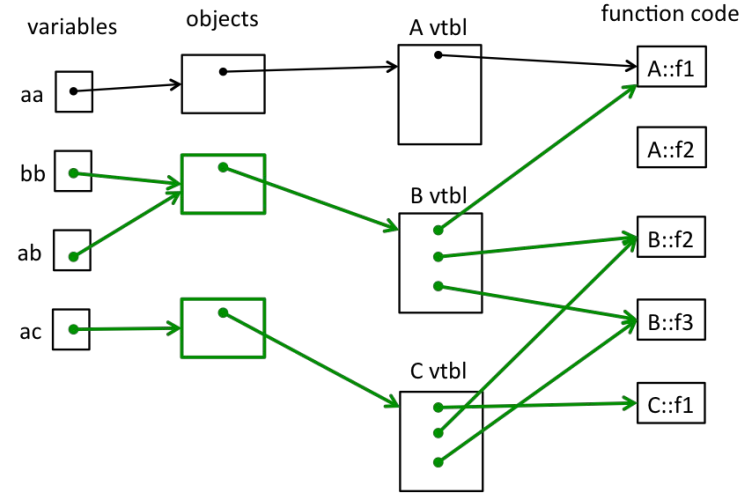
Exercise 2 Solution (output)

```
#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
```

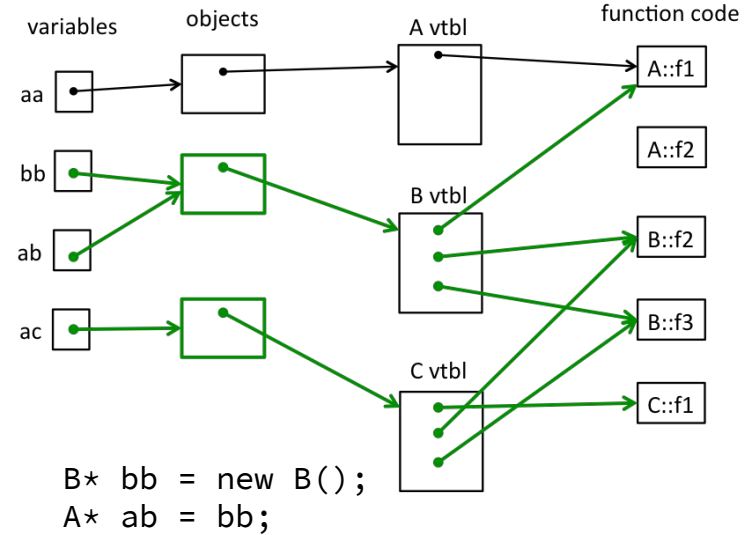
Exercise 2 Solution (output)

```
#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; }
};
```



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

B::f2	A::f2	B::f2	A::f2
----	----	----	----
B::f2	B::f2	A::f2	A::f2
✓	✗	⏪	⏩
yes	no	go slower	go faster

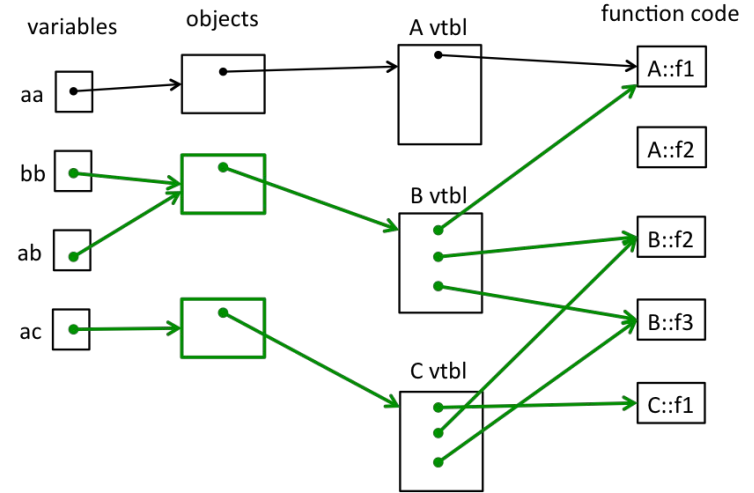
Exercise 2 Solution (output)

```
#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::f3
----
A::f2
```

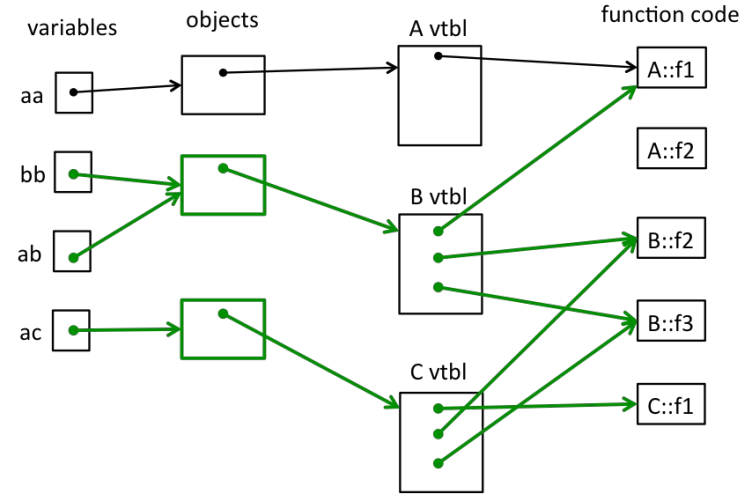
Exercise 2 Solution (output)

```
#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();

B::f2	A::f2	A::f2	B::f2
A::f1	A::f1	C::f1	C::f1
B::f3	B::f3	B::f3	B::f3
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
yes	no	go slower	go faster

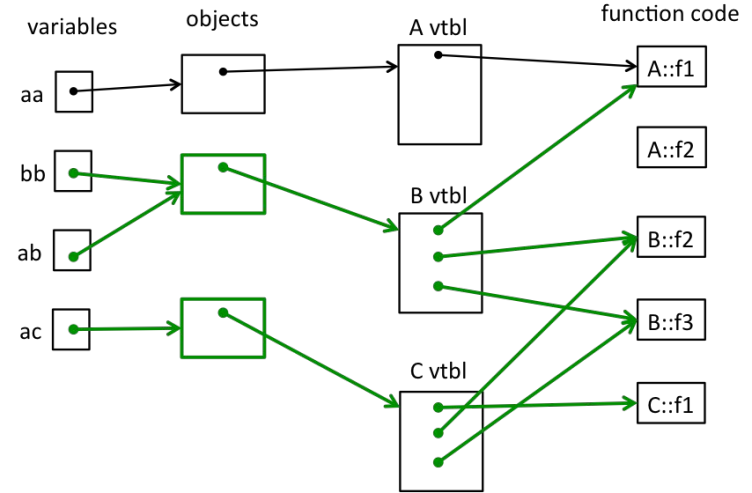
Exercise 2 Solution (output)

```
#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
```

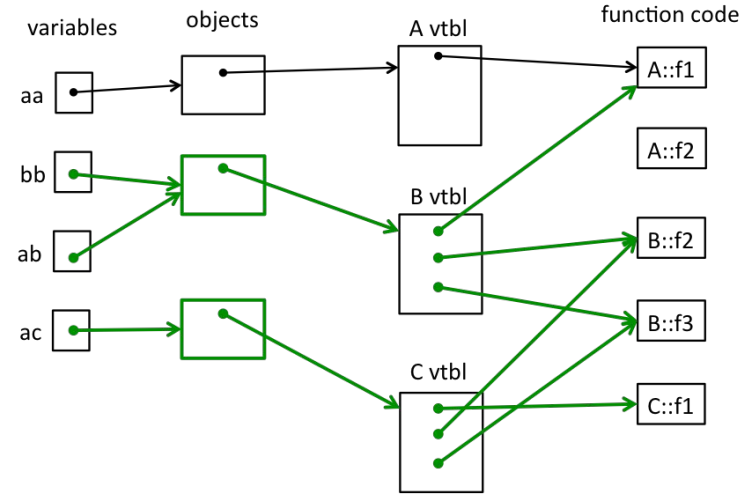
Exercise 2 Solution (output)

```
#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();
```

B::f2	A::f2	A::f2	B::f2
A::f1	C::f1	A::f1	C::f1

yes no go slower go faster

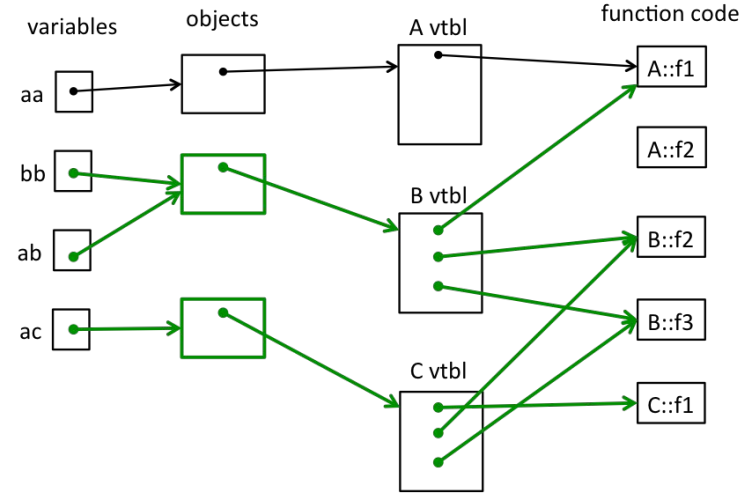
Exercise 2 Solution (output)

```
#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
```

**Any Questions on
Inheritance & VTable?**

**TAs: Remember to plan for Flex
Time!!!**

**Thanks for coming
to section!**