



pollev.com/cse333x



About how long did Exercise 14 take you?

- A. [0, 2) hours
- B. [2, 4) hours
- C. [4, 6) hours
- D. [6, 8) hours
- E. 8+ Hours
- F. I didn't submit / I prefer not to say

Systems Programming

C++ Inheritance I

Guest Lecturer:

Jen Xu

Teaching Assistants:

Ally Tribble

Blake Diaz

Connor Olson

Grace Zhou

Jackson Kent

Janani Raghavan

Jen Xu

Jessie Sun

Jonathan Nister

Mendel Carroll

Rose Maresh

Violet Monserate

Relevant Course Information

- ❖ Mid-quarter survey responses will be addressed once Justin and Amber return
- ❖ Exercise 13 released Friday
 - C++ and inheritance (requires both lectures)
- ❖ Homework 3 is due next Thursday (2/26)
 - Get started early!
 - Videos for overview and demo and file debugging
 - Partner sign-ups close at end of Thursday (2/19)
- ❖ Midterm grades are out!

Overview of Next Two Lectures

❖ C++ inheritance

- **Review of basic idea** (pretty much the same as in Java)
- What's different in C++ (compared to Java)
 - **Static vs. dynamic dispatch – virtual functions and vtables** (optional)
 - Pure virtual functions, abstract classes, why no Java “interfaces”
 - Assignment slicing, using class hierarchies with STL
- Casts in C++

❖ Reference: *C++ Primer*, Chapter 15

Lecture Outline (1/3)

- ❖ **Inheritance Motivation & C++ Syntax**
- ❖ Polymorphism & Dynamic Dispatch
- ❖ Virtual Tables & Virtual Table Pointers

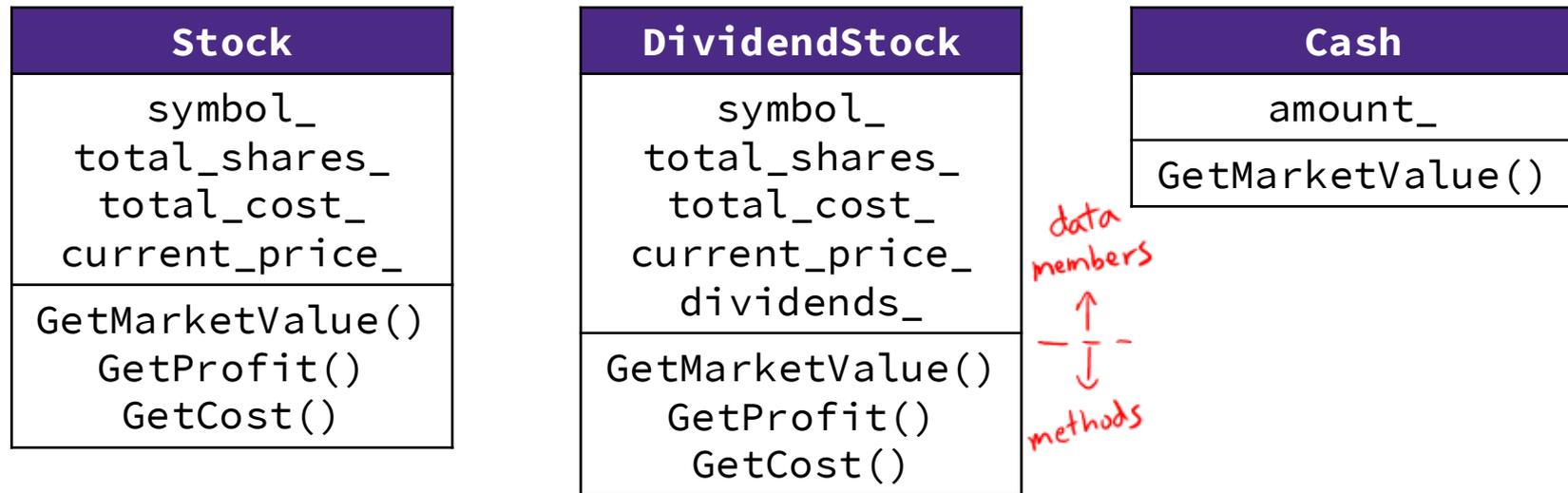
Stock Portfolio Example

- ❖ A portfolio represents a person's financial investments
 - Each *asset* has a cost (*i.e.*, how much was paid for it) and a market value (*i.e.*, how much it is worth)
 - The difference between the cost and market value is the *profit* (or loss)
 - Different assets compute market value in different ways
 - A **stock** that you own has a ticker symbol (*e.g.*, "GOOG"), a number of shares, share price paid, and current share price
 - A **dividend stock** is a stock that also has dividend payments
 - **Cash** is an asset that never incurs a profit or loss

(Credit: thanks to Marty Stepp for this example)

Design Without Inheritance

- ❖ One class per asset type:



- Redundant!
- Cannot treat multiple investments together
 - e.g., can't have an array or vector of different assets

↑ must be of same type

- ❖ See sample code in `initial/` directory

Inheritance Terminology

- ❖ A parent-child “is-a” relationship between classes
 - A child (**derived class**) extends a parent (**base class**)

- ❖ Terminology:

*“higher” on heirarchy
subset of subclass*

*“lower” on heirarchy
superset of superclass*

Java	C++
Superclass	<u>Base Class</u>
Subclass	<u>Derived Class</u>

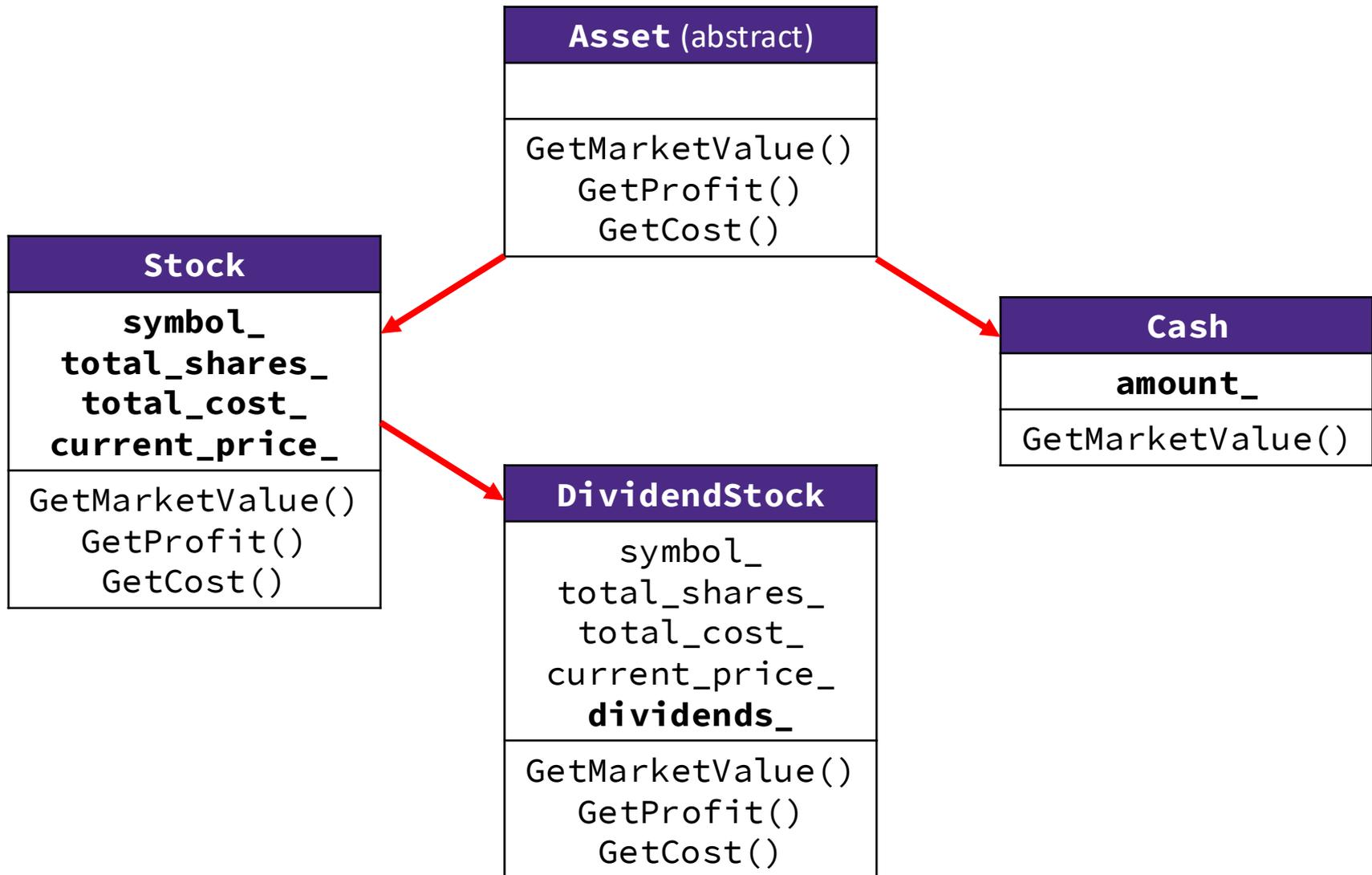
- Mean the same things. You’ll hear both.

Inheritance Benefits

- ❖ A parent-child “is-a” relationship between classes
 - A child (**derived class**) extends a parent (**base class**)

- ❖ Benefits:
 - Code reuse
 - Children can automatically inherit code from parents
 - Polymorphism
 - Ability to redefine existing behavior but preserve the interface
 - Children can override the behavior of the parent
 - Others can make calls on objects without knowing which part of the inheritance tree it is in
 - Extensibility
 - Children can add behavior

Design With Inheritance



Access Modifiers (like Java)

- ❖ `public`: visible to all other classes
- ❖ `protected`: visible to current class and its *derived* classes
- ❖ `private`: visible only to the current class

- ❖ Use `protected` for class members only when
 - Class is designed to be extended by derived classes
 - Derived classes must have access but clients should not be allowed

Class Derivation List

- ❖ Comma-separated list of classes to inherit from:

```
#include "BaseClass.h"

class Name : public BaseClass {
    ...
};
```



- Focus on **single inheritance**, but *multiple inheritance* possible
: public Base1, public Base2 {
- ❖ Almost always you will want **public inheritance**
 - Acts like extends does in Java
 - Any member that is non-private in the base class is the same in the derived class; both *interface and implementation inheritance*
 - ★ Except that constructors, destructors, copy constructor, and assignment operator are *never* inherited

Stocks Classes Relationship (1/2)

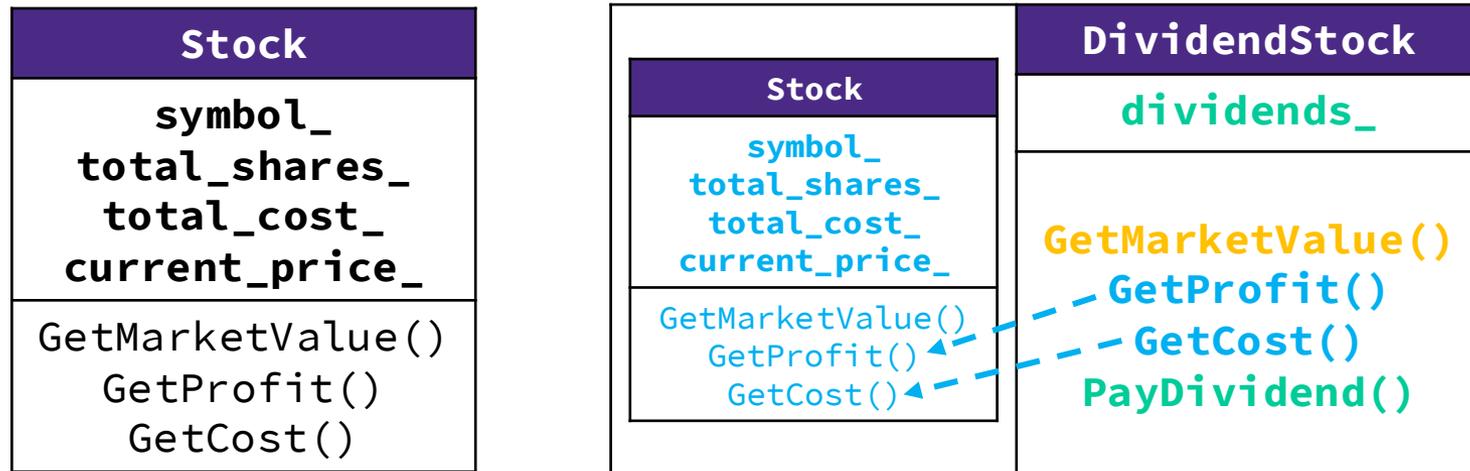
Stock
symbol_ total_shares_ total_cost_ current_price_
GetMarketValue() GetProfit() GetCost()

BASE

DividendStock
symbol_ total_shares_ total_cost_ current_price_ dividends_
GetMarketValue() GetProfit() GetCost()

DERIVED

Stocks Classes Relationship (2/2)



- ❖ A derived class:
 - **Inherits** the behavior and state (specification) of the base class
 - **Overrides** some of the base class' member functions (opt.)
 - **Extends** the base class with new member functions, variables (opt.)

Lecture Outline (2/3)

- ❖ Inheritance Motivation & C++ Syntax
- ❖ **Polymorphism & Dynamic Dispatch**
- ❖ Virtual Tables & Virtual Table Pointers

Polymorphism in C++

- ❖ In Java: `PromisedType var = new ActualType();`
 - `var` is a reference (different term than C++ reference) to an object of `ActualType` on the Heap
 - `ActualType` must be the same class or a subclass of `PromisedType`
- ❖ In C++: `PromisedType* var_p = new ActualType();`
 - `var_p` is a *pointer* to an object of `ActualType` on the Heap
 - `ActualType` must be the same or a derived class of `PromisedType`
 - (also works with references)
- ✳ `PromisedType` defines the *interface* (i.e., what can be called on `var_p`), but `ActualType` may determine which *version* gets invoked

Dynamic Dispatch (like Java)

- ❖ Usually, when a derived function is available for an object, we want the derived function to be invoked
 - This requires a *run time* decision of what code to invoke
- ❖ A member function invoked on an object should be the *most-derived function* accessible to the object's visible type
 - Can determine what to invoke from the *object* itself

- ❖ Example:

- `void PrintStock(Stock* s) { s->Print(); }`

- Calls the appropriate `Print()` without knowing the actual type of `*s`, other than it is some sort of `Stock`

Stock?
DividendStock?



Dynamic Dispatch Example (1/2)

- ❖ When a member function is invoked on an object:
 - The *most-derived function* accessible to the object's visible type is invoked (decided at run time based on actual type of the object)

```
double DividendStock::GetMarketValue() const {
    return get_shares() * get_share_price() + dividends_;
}

double "DividendStock"::GetProfit() const { // inherited
    return GetMarketValue() - GetCost();
}
      ↑ should invoke DividendStock::GetMarketValue()
                                           DividendStock.cc
```

inherited from Stock

```
double Stock::GetMarketValue() const {
    return get_shares() * get_share_price();
}

double Stock::GetProfit() const {
    return GetMarketValue() - GetCost();
}
```

Stock.cc

Dynamic Dispatch Example (2/2)

```
#include "Stock.h"
#include "DividendStock.h"

DividendStock dividend;
DividendStock* ds = &dividend;
Stock* s = &dividend;    // why is this allowed?

// Invokes DividendStock::GetMarketValue()
ds->GetMarketValue();

// Invokes DividendStock::GetMarketValue()
s->GetMarketValue();

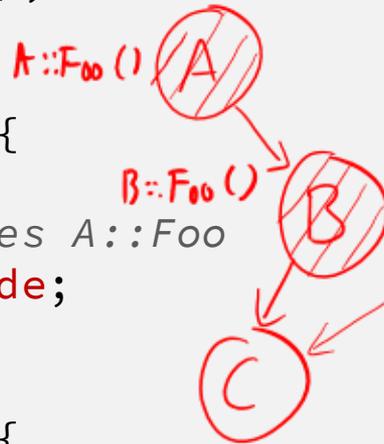
// invokes Stock::GetProfit(), since that method is inherited.
// Stock::GetProfit() invokes DividendStock::GetMarketValue(),
// since that is the most-derived accessible function.
s->GetProfit();
```

Requesting Dynamic Dispatch (C++)

- ❖ Prefix the member function declaration with the `virtual` keyword
 - Derived/child functions don't need to repeat `virtual`, but was traditionally good style to do so
 - This is how method calls work in Java (no virtual keyword needed)
 - You almost always want functions to be virtual
- ❖ `override` keyword (C++11)
 - Tells compiler this method should be overriding an inherited virtual function – *always* use if available
 - Prevents overloading vs. overriding bugs
- ❖ Both of these are technically *optional* in derived classes
 - Be consistent and follow local conventions (Google Style Guide says no `virtual` if `override`)

Most-Derived

```
class A {  
public:  
    // Foo will use dynamic dispatch  
    virtual void Foo();  
};  
  
class B : public A {  
public:  
    // B::Foo overrides A::Foo  
    void Foo() override;  
};  
  
class C : public B {  
    // C inherits B::Foo()  
};
```



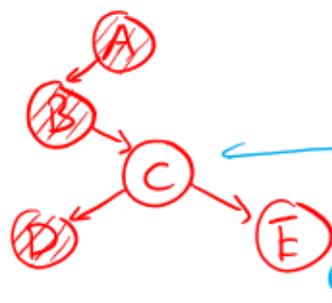
```
void Bar() {  
    A* a_ptr;  
    C c;  
  
    a_ptr = &c;  
  
    // Whose Foo() is called?  
    a_ptr->Foo(); // B::Foo()  
}
```

Poll Everywhere

pollev.com/cse333x



Whose **Foo()** is called?



```
void Bar() {
    A* a_ptr;
    C c;
    E e;

    // Q1:
    a_ptr = &c;
    a_ptr->Foo();
    // B::Foo()

    // Q2:
    a_ptr = &e;
    a_ptr->Foo();
    // B::Foo()
}
```

```
class A {
public:
    virtual void Foo();
};

class B : public A {
public:
    void Foo() override;
};

class C : public B {
};

class D : public C {
public:
    void Foo() override;
};

class E : public C {
};
```

Q1 Q2

A. A B

B. A D

C. B B

D. B D

E. We're lost...

Lecture Outline (3/3)

- ❖ Inheritance Motivation & C++ Syntax
- ❖ Polymorphism & Dynamic Dispatch
- ❖ **Virtual Tables & Virtual Table Pointers**

How Can This Possibly Work?

- ❖ The compiler produces `Stock.o` from *just* `Stock.cc`
 - It doesn't know that `DividendStock` exists during this process
 - So then how does the emitted code know to call `Stock::GetMarketValue()` or `DividendStock::GetMarketValue()` or something else that might not exist yet?
 - ***Function pointers!!!***

Stock.h

```
virtual double Stock::GetMarketValue() const;  
virtual double Stock::GetProfit() const;
```

```
double Stock::GetMarketValue() const {  
    return get_shares() * get_share_price();  
}
```

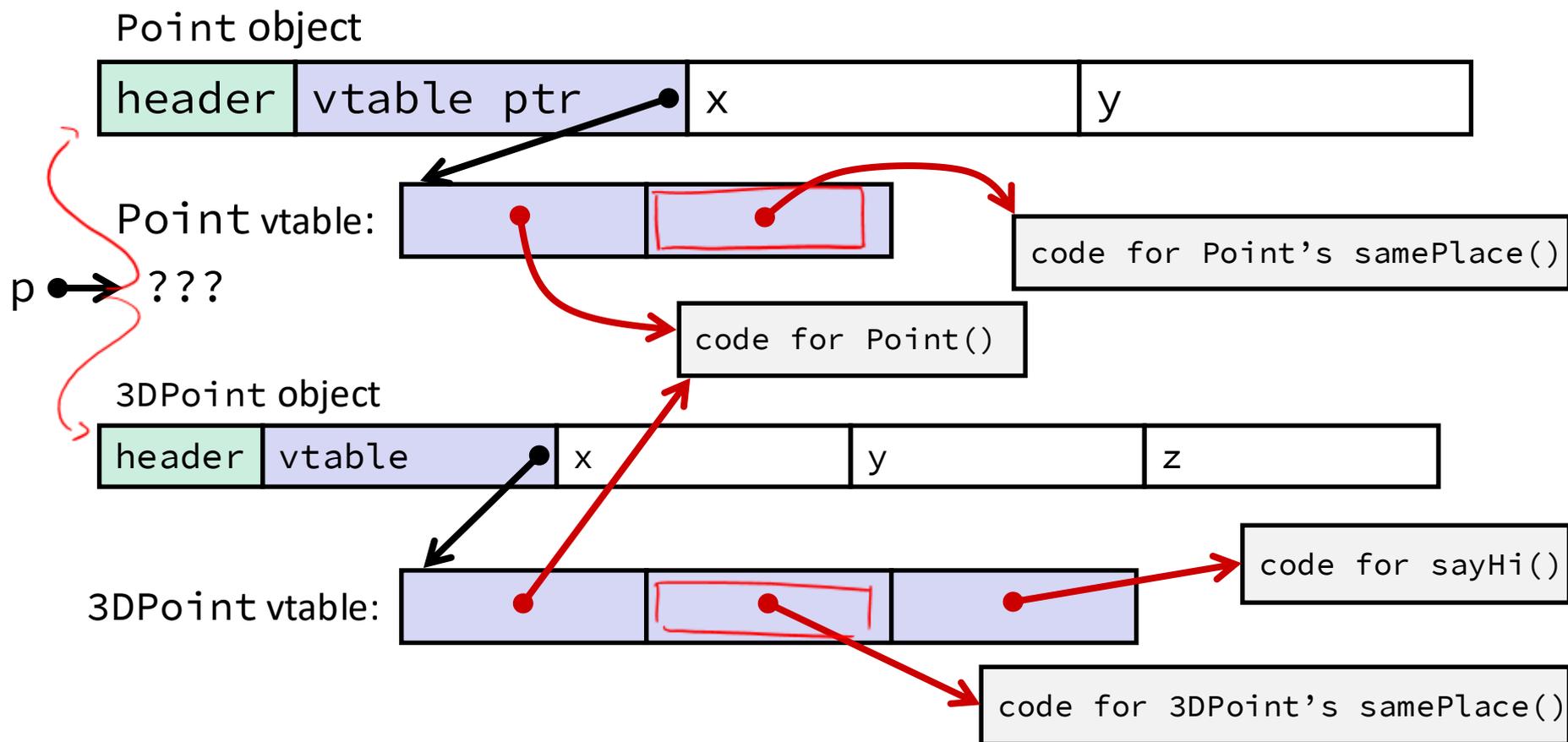
```
double Stock::GetProfit() const {  
    return GetMarketValue() - GetCost();  
}
```

Stock.cc

vtables and the vptr

- ❖ If a class contains *any* virtual methods, the compiler emits:
 - A (single) virtual function table (**vtable**) for the class (1 per class)
 - Contains a function pointer for each virtual method in the class
 - The pointers in the vtable point to the most-derived function for that class
 - A virtual table pointer (**vptr**) for each object instance (1 per object)
 - A pointer to a virtual table as a “hidden” member variable
 - When the object’s constructor is invoked, the vptr is initialized to point to the vtable for the object’s class
 - Thus, the vptr “remembers” what class the object is

351 Throwback: Dynamic Dispatch



Java:

```
Point p = ???;
return p.samePlace(q);
```

could be Point or 3DPoint

C pseudo-translation:

```
// works regardless of what p is
return p->vtable[1](p, q);
```

vtable/vptr Example: Dispatch

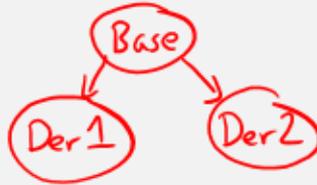
```

class Base {
public:
    virtual void F1();
    virtual void F2();
};

class Der1 : public Base {
public:
    void F1() override;
};

class Der2 : public Base {
public:
    void F2() override;
};

```



```

Base b;
Der1 d1;
Der2 d2;

Base* b0ptr = &b;
Base* b1ptr = &d1;
Base* b2ptr = &d2;

b0ptr->F1(); // Base::F1()
b0ptr->F2(); // Base::F2()

b1ptr->F1(); // Der1::F1()
b1ptr->F2(); // Base::F2()

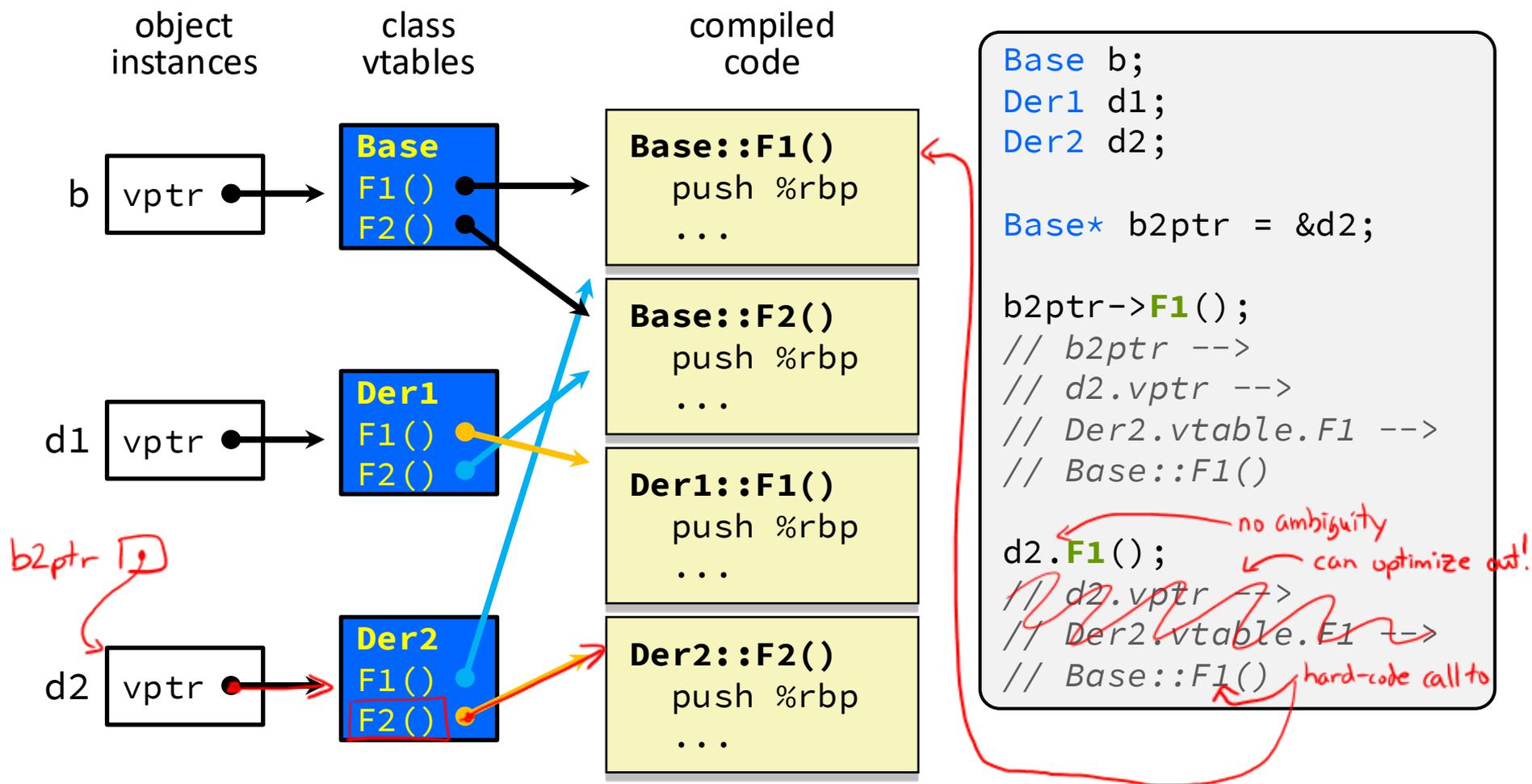
b2ptr->F1(); // Base::F1()
b2ptr->F2(); // Der2::F2()

d2.F1(); // Base::F1()

```

difference?

vtable/vptr Example: Tables



Let's Look at Some Actual Code

- ❖ Let's examine the following code using objdump
 - `g++ -Wall -g -std=c++17 -o vtable vtable.cc`
 - `objdump -CDS vtable > vtable.d`

vtable.cc

```
class Base {
public:
    virtual void f1();
    virtual void f2();
};

class Der1 : public Base {
public:
    void f1() override;
};

int main(int argc, char** argv) {
    Der1 d1;
    Base* bptr = &d1;
    bptr->f1(); // done via indirect jump on vtable entry
    d1.f1();   // done via hard-coded callq
}
```