C++ Inheritance I

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

Poll Everywhere

pollev.com/cse333

About how long did Exercise 12a take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I'm not done yet / I prefer not to say

Administrivia

- Exercise 13 (Skip List) extended until tomorrow

- Exercise 14 (Inheritance) still assigned for today, due Wed

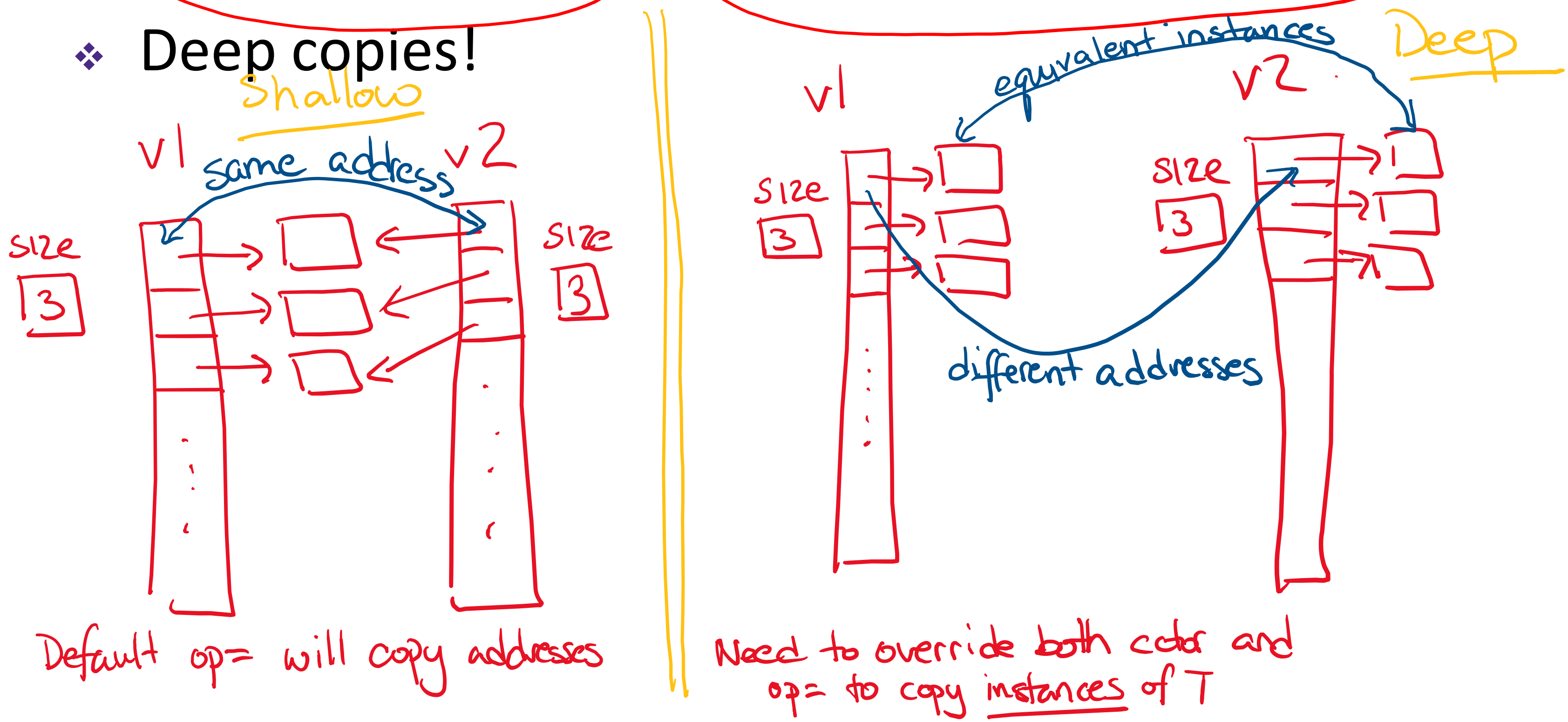
- Midterm: Scores/feedback published
  - Some statistics:
    - Mean: 79% (89 pts), Standard Deviation: 12% (13 pts)
  - Regrade Requests open today
    - Submit regrades for individual parts, after looking at sample solution!
  - Remember! The midterm is a tool to check your understanding, NOT an indicator of your ability to do systems programming!
    - Midterm: 15% of final grade (Final: 20%, EX + HW: 60%)

Lecture Outline

- **Midterm Misunderstandings**
- C++ Inheritance
  - Review of basic idea
  - Dynamic Dispatch, Conceptually
  - Dynamic Dispatch, Implementation: vtables and vptr

Midterm Misunderstandings

- `T *contents_` vs `T* contents_[64]`
- Deep copies!



Lecture Outline

- Midterm Misunderstandings
- C++ Inheritance
  - **Review of basic idea**
  - Dynamic Dispatch, Conceptually
  - Dynamic Dispatch, Implementation: vtables and vptr

Stock Portfolio Example

- A portfolio represents a person's 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, which contributes to your profit
    - **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:

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

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

Cash
amount_
GetMarketValue()

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

- See sample code: `initial/`

Inheritance

- An "is-a" relationship: a child "is-a" parent
  - A child (derived class) extends a parent (base class)

- Terminology:

Java	C++
Superclass	Base Class
Subclass	Derived Class

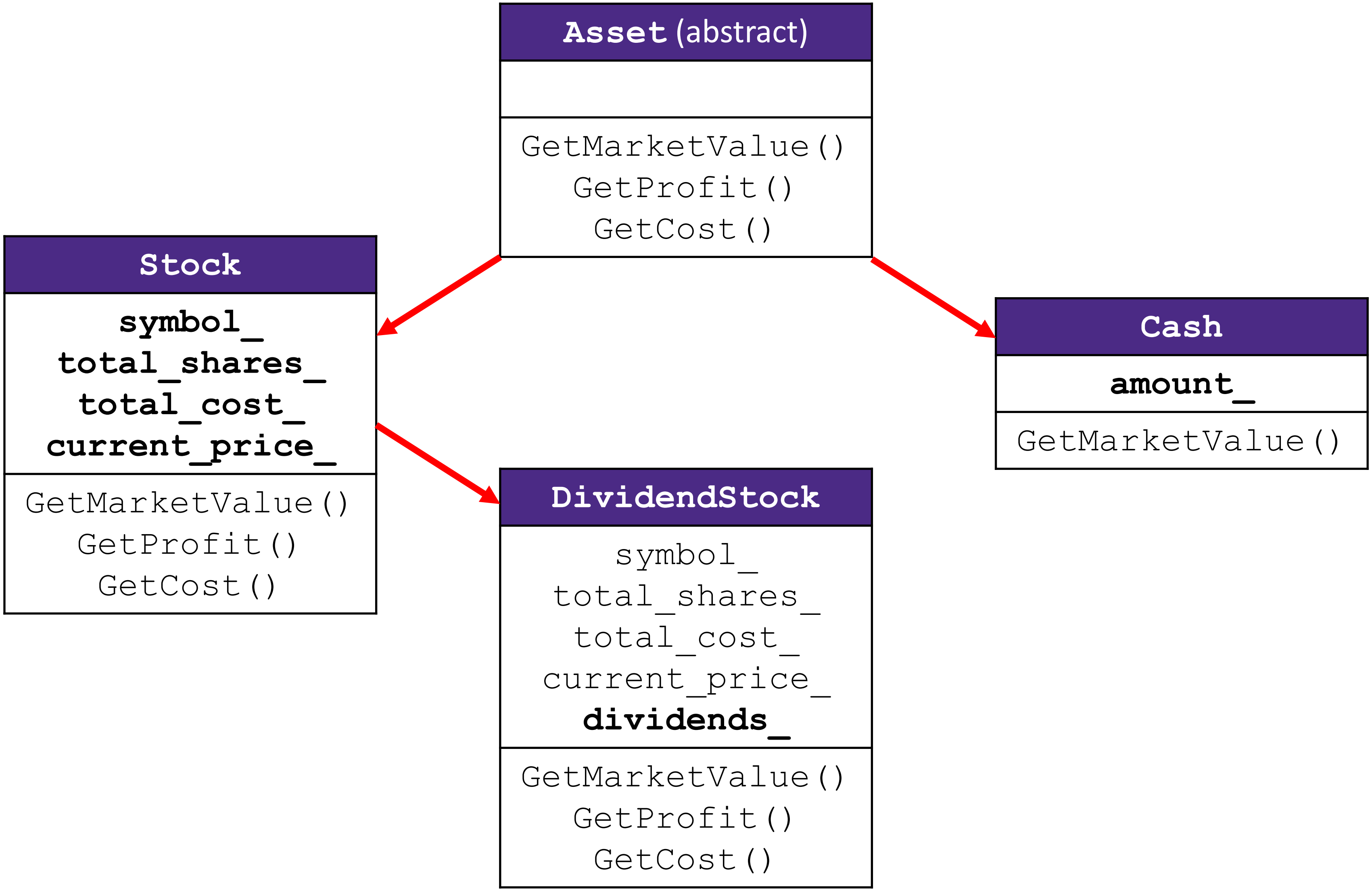
  - Mean the same things. You'll hear both.

Inheritance

- An "is-a" relationship: a child "is-a" parent
  - 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



See sample code: `inherit/`

Like Java: Access Modifiers

- `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 subclasses
  - Subclasses 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

- 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

Back to Stocks

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

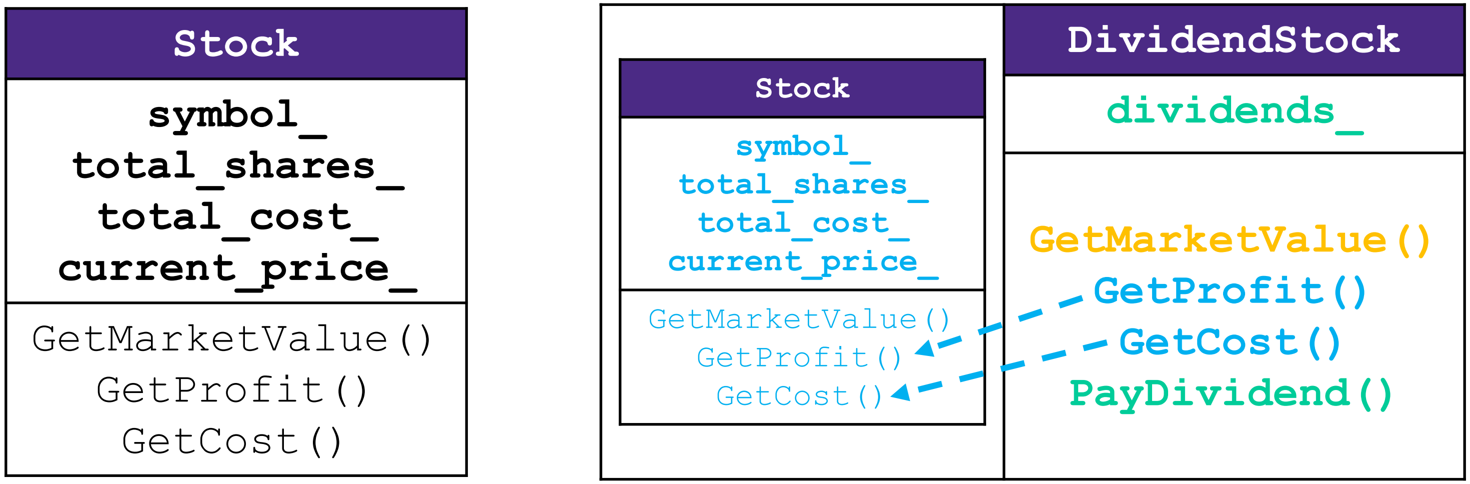
Polymorphism in C++

- 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

Back to Stocks



- 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

- Midterm Misunderstandings
- C++ Inheritance
  - Review of basic idea
  - **Dynamic Dispatch, Conceptually**
  - Dynamic Dispatch, Implementation: vtables and vptr

Most-Derived

```
class A {
 public:
  // Foo will use dynamic dispatch
  virtual void Foo();
};

class B : public A {
 public:
  // B::Foo overrides A::Foo
  virtual void Foo();
};

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();
}
```

Dynamic Dispatch (similarities to 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`

Dynamic Dispatch (C++-specific)

- Prefix the "highest" member function declaration with the `virtual` keyword
  - This is how method calls work in Java (no virtual keyword needed)
  - Derived/child functions will be "virtual", so repeating `virtual` declaration is technically *optional*
    - Traditionally good style to do so!

- Derived/child functions should use `override`
  - Tells compiler this method should be overriding an inherited virtual function – *always* use if available (added in C++11)
  - Prevents overloading vs. overriding bugs

Dynamic Dispatch Example

- 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 {  // not actually here;
  return GetMarketValue() - GetCost();       // inherited from Stock
}
```

DividendStock.cc

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

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

Stock.cc

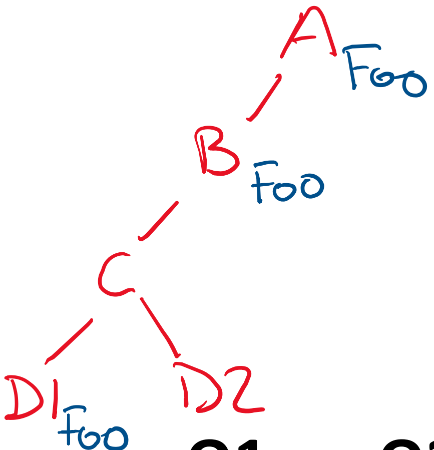
Dynamic Dispatch Example

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

DividendStock dividend;
DividendStock *s = &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();
```

Poll Everywhere

pollev.com/cse333

- Whose `Foo()` is called?



	Q1	Q2
A.	A	B
B.	A	D2
C.	B	B
D.	B	D2

E. I'm not sure…

```
void Bar() {
  A *a_ptr;

  // Q1:
  a_ptr = new C;
  a_ptr->Foo();

  // Q2:
  a_ptr = new E;
  a_ptr->Foo();
}
```

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

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

class C : public B {
};

class D1 : public C {
 public:
  virtual void Foo();
};

class D2 : public C {
};
```

`virtual` is "sticky"

- If `X::f()` is declared virtual, then a vtable will be created for class `X` and for *all* of its subclasses
  - The vtables will include function pointers for (the correct) `f`

- `f()` will be called using dynamic dispatch even if overridden in a derived class without the `virtual` keyword
  - Good style to help the reader *and avoid bugs* by using `override`
    - Style guide controversy, if you use `override` should you use `virtual` in derived classes? Recent style guides say just use `override`, but you'll sometimes see both, particularly in older code

Lecture Outline

- Midterm Misunderstandings
- C++ Inheritance
  - Review of Basic Idea
  - Dynamic Dispatch, Conceptually
  - **Dynamic Dispatch, Implementation: vtables and vptr**

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;
```

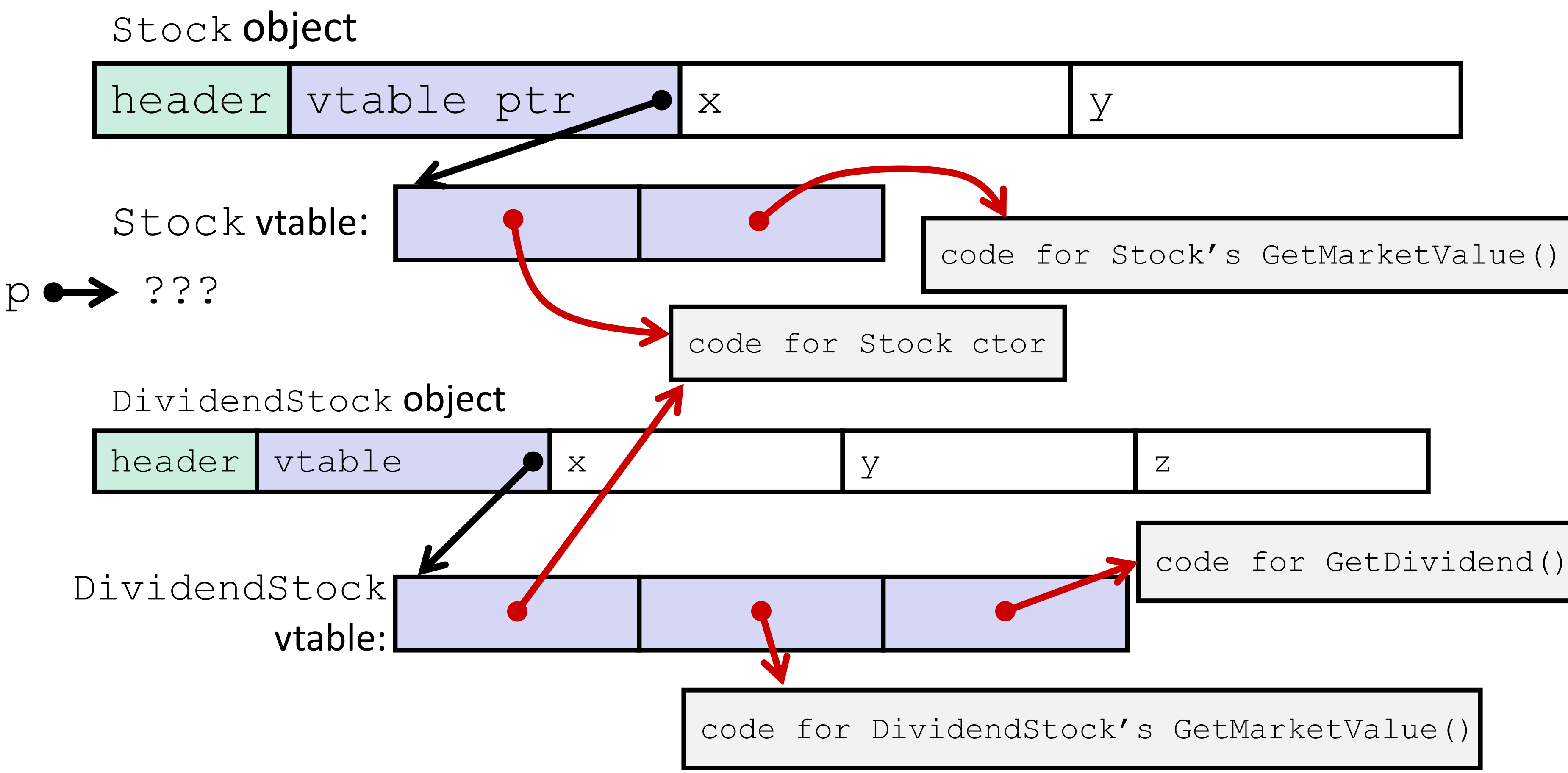
```
double Stock::GetMarketValue() const {
  return get_shares() * get_share_price();
}

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*
    - 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*
    - 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:

```
Stock s = ???;
return s.GetMarketValue();
```

C pseudo-translation:

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

vtable/vptr Example

```
class Base {
 public:
  virtual void func1();
  virtual void func2();
};

class Der1 : public Base {
 public:
  virtual void func1();
};

class Der2 : public Base {
 public:
  virtual void func2();
};
```

```
Base b;
Der1 d1;
Der2 d2;

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

b0ptr->func1();   //
b0ptr->func2();   //

b1ptr->func1();   //
b1ptr->func2();   //

d2.func1();       //
b2ptr->func1();   //
b2ptr->func2();   //
```

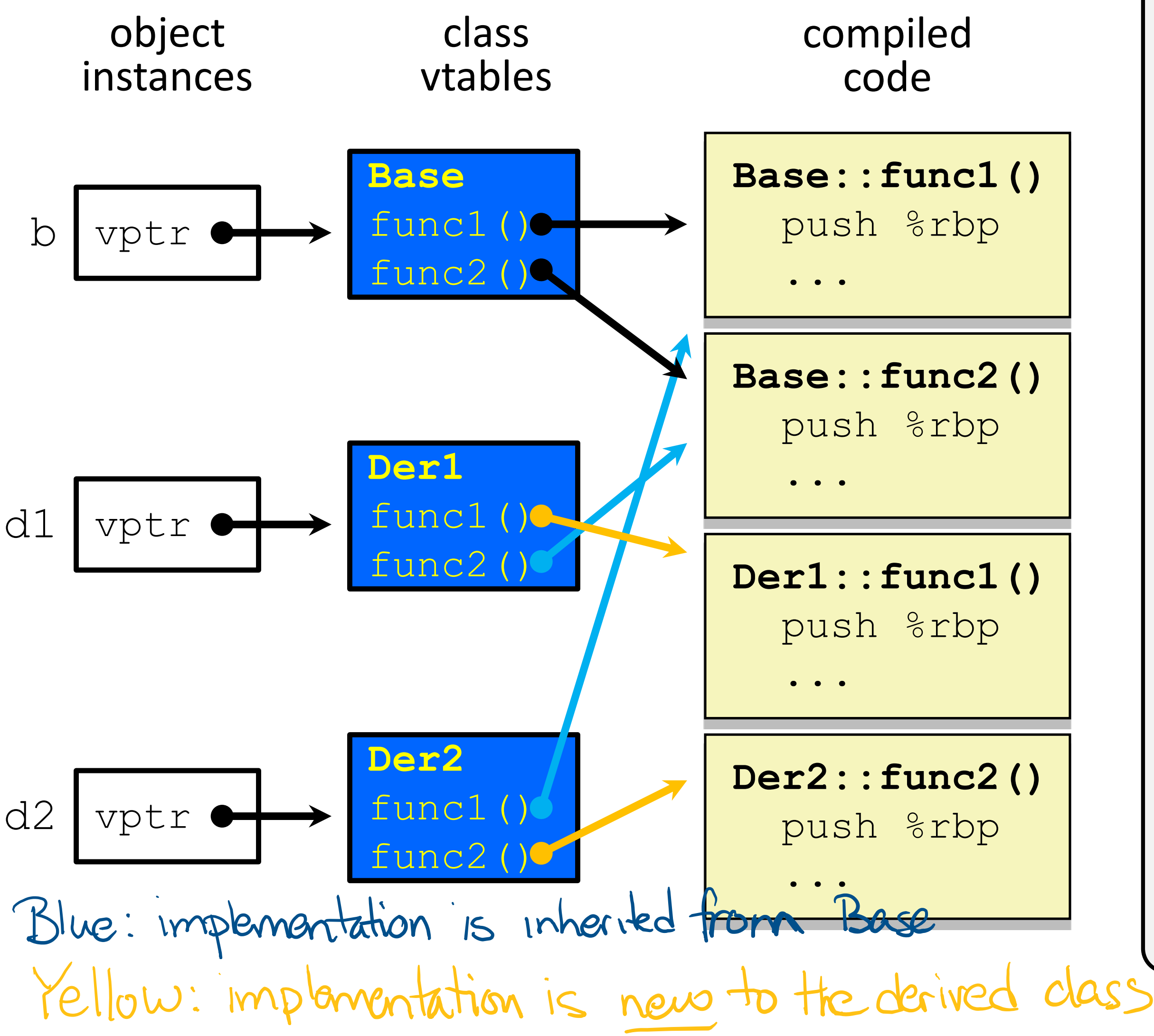
vtable/vptr Example

Base: func1, func2
Der1: func1
Der2: func2

```
class Base {
 public:
  virtual void func1();
  virtual void func2();
};

class Der1 : public Base {
 public:
  virtual void func1();
};

class Der2 : public Base {
 public:
  virtual void func2();
};
```

```
Base b;
Der1 d1;
Der2 d2;

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

b0ptr->func1();   // Base
b0ptr->func2();   // Base

b1ptr->func1();   // Der1
b1ptr->func2();   // Base

d2.func1();       // Base
b2ptr->func1();   // Base
b2ptr->func2();   // Der2
```

vtable/vptr Example




```
Base b;
Der1 d1;
Der2 d2;

Base *bptr = &d1;

bptr->func1();
// bptr -->
// d1.vptr -->
// Der1.vtable.func1
//    -->
// Base::func1()

bptr = &d2;

bptr->func1();
// bptr -->
// d2.vptr -->
// Der2.vtable.f1 -->
// Base::f1()
```

Let's Look at Some Actual Code

- Let's examine the following code using `objdump`
  - `g++ -Wall -g -std=c++11 -o vtable vtable.cc`
  - `objdump -CDS vtable > vtable.d`

vtable.cc

```
class Base {
 public:
  virtual void func1();
  virtual void func2();
};

class Der1 : public Base {
 public:
  virtual void func1();
};

int main(int argc, char **argv) {
  Der1 d1;
  d1.func1();
  Base *bptr = &d1;
  bptr->func1();
}
```