Poll Everywhere

pollev.com/naomila

Which of the following is *not* true?

A. C++ classes and structs are effectively the same things
B. C syscalls and standard library functions are directly callable from C++
C. C++ can have multiple functions defined with the same name
D. In C++ there is no difference between declaration and definition
E. Templates are "compiled out" by the time the program is running
F. These are all true, you *rotten sneak-artist*

CSE333 Systems Programming
C++ Inheritance I

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

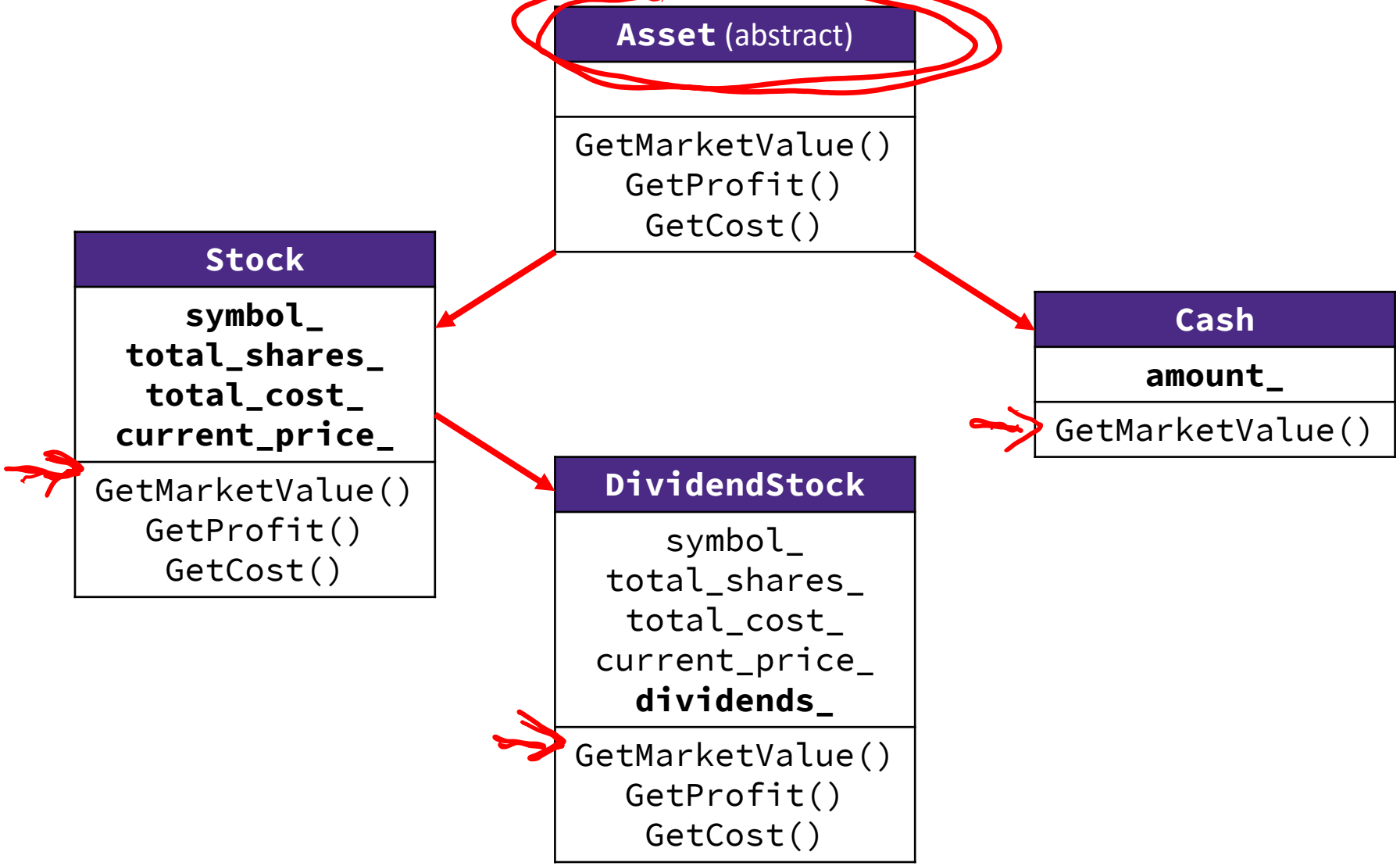
Administrivia

- Ex8 due on Wednesday
- Get started on HW3!
- Midterms to be graded soon

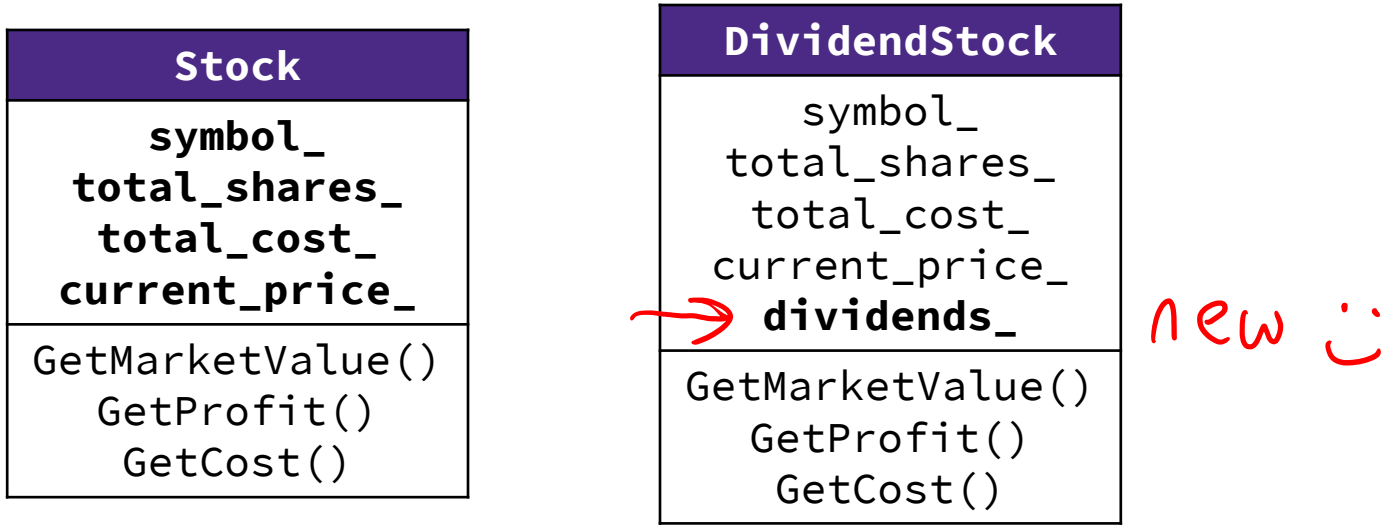
Lecture Outline

- **Brief review: toy example**
- Polymorphism & Method Dispatch
- Under the hood: Virtual Tables (vtables)

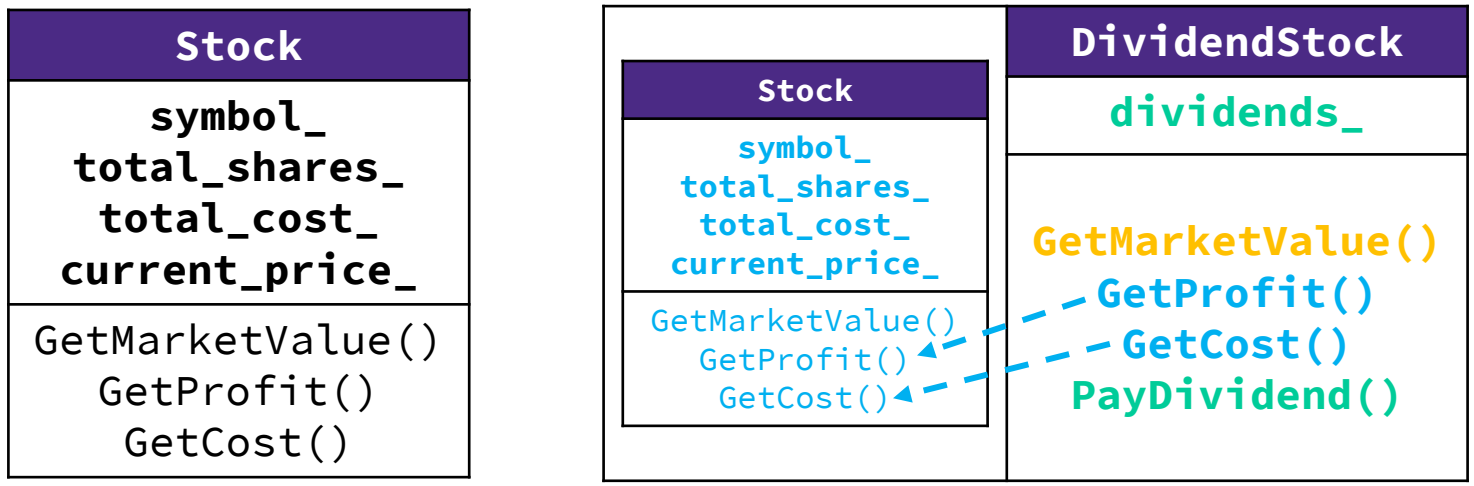
Toy example: financial portfolio



Stocks Classes Relationship (1/2)



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

- Brief review: toy example
- **Polymorphism & Method Dispatch**
- Under the hood: Virtual Tables (vtables)

Polymorphism in C++

- In Java: `PromisedType var = new ActualType();`
  - `var` is a Java reference (different meaning 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

Method Dispatch

- Let's say our class interface has a method implemented by some children but not others:

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

- Which version of `Print()` do we want to call ("dispatch to") ?
  C++ gives the developer of the `Stock` class two options:

1. The `Print()` defined by the type visible in the code (`Stock::Print`)

   This is called **static dispatch**. It is the **default behavior** in C++.

2. The `Print()` defined by the type of the object in memory (maybe `Stock::Print`, maybe `DividendStock::Print`, maybe some fancy future subclass)

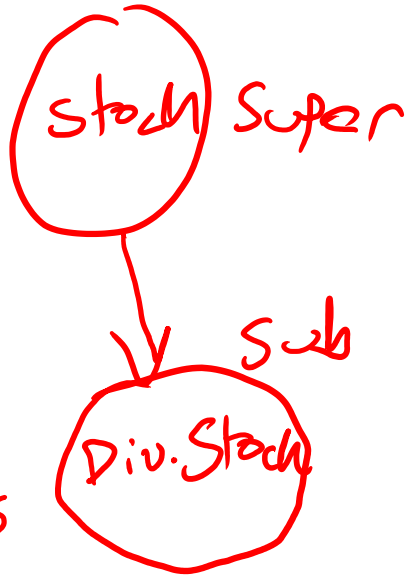
   This is called **dynamic dispatch**. It is **opt-in behavior** on the part of the **class's author**.

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 *runtime* 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

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();
}
```

DividendStock.cc

```
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();
```

b/c it's dynamic

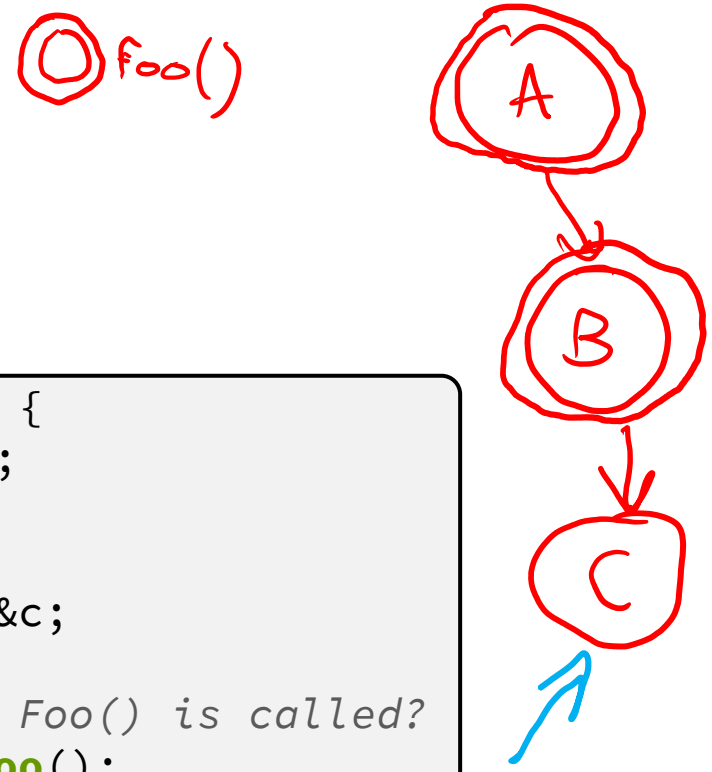
Requesting Dynamic Dispatch (C++)

- 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* if the function is already virtual in a superclass
  - 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();
}
```

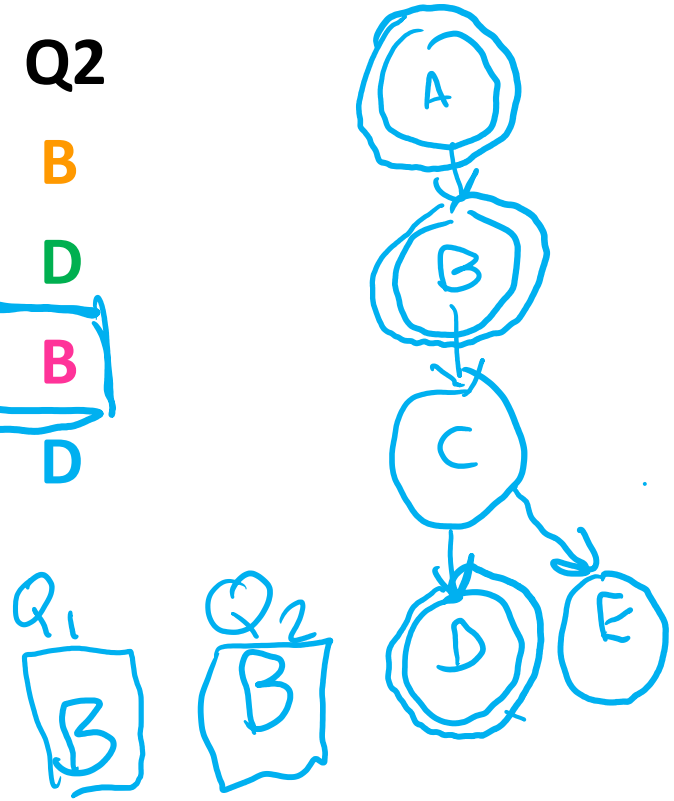


Poll Everywhere

pollev.com/naomila

Whose Foo( ) is called?

	Q1	Q2
A.	A	B
B.	A	D
C.	B	B
D.	B	D
E.	Nah…	




```
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 {
};
```

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

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

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

Lecture Outline

- Brief review: toy example
- Polymorphism & Method Dispatch
- **Under the hood: Virtual Tables (vtables)**

How Can Dynamic Dispatch 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

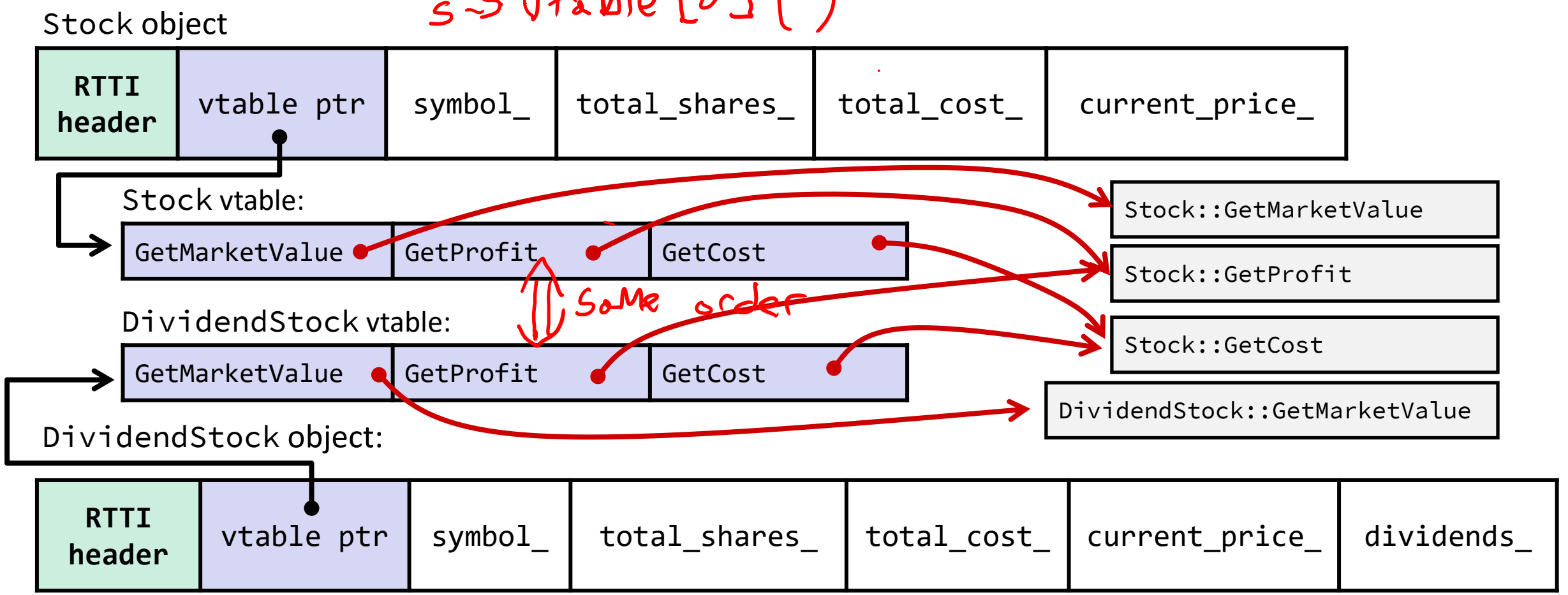
- If a class contains *any* virtual methods, the compiler emits:
  - A (single) virtual function table (vtable) for *the class* ← one per class type
    - Contains a function pointer for each virtual method in the class
    - The pointers in the vtable point to the most-derived function for that class
  - A virtual table pointer (vptr) for *each object instance* ← one per class 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

C++ classes in memory

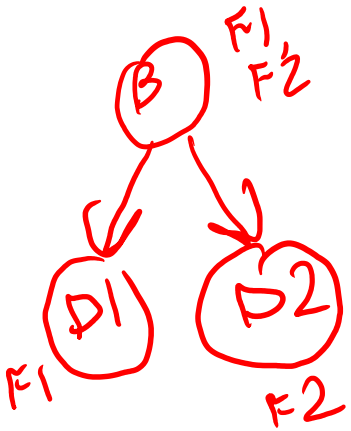
RTTI ⇒ Run-time type information

```
Stock *s = ???
std::cout << s->GetMarketValue()
```

s -> vtable[0]()



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
b0ptr->F2();  // Base

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

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

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

`virtual` is "sticky" downward

- 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

What happens if we omit "virtual"?

- By default, without `virtual`, methods are dispatched *statically*
  - At compile time, the compiler writes in a `call` to the address of the class' method in the `.text` segment
    - Based on the compile-time visible type of the callee
  - This is *different* than Java

```
class Derived : public Base { ... };

int main(int argc, char** argv) {
  Derived d;
  Derived* dp = &d;
  Base* bp = &d;
  dp->Foo();
  bp->Foo();
  return EXIT_SUCCESS;
}
```

Static Dispatch Example

- Removed `virtual` on methods:

Stock.h

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

defined in Stock & DivStock
defined in Stock, inherited by DivStock, calls getMarketValue()

```
DividendStock dividend;
DividendStock* ds = &dividend;
Stock* s = &dividend;

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

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

// invokes Stock::GetProfit().
// Stock::GetProfit() invokes Stock::GetMarketValue().
s->GetProfit();

// invokes Stock::GetProfit(), since that method is inherited.
// Stock::GetProfit() invokes Stock::GetMarketValue().
ds->GetProfit();
```

Why Not Always Use `virtual`?

- Two (fairly uncommon) reasons:
  - Efficiency:
    - Non-virtual function calls are a tiny bit faster (no indirect lookup)
    - A class with no virtual functions has objects without a `vptr` field
  - Control:
    - If `F()` calls `G()` in class X and G is not virtual, we're guaranteed to call `X::G()` and not `G()` in some subclass
      - Particularly useful for framework design
- In Java, all methods are virtual, except `static` class methods, which aren't associated with objects
- In C++ and C#, you can pick what you want
  - Omitting `virtual` can cause obscure bugs
  - (Most of the time, you want member function to be `virtual`)