What do you think?

Name that Constructor!

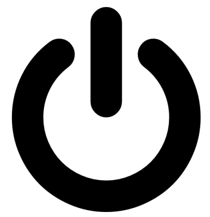
What type of constructors do you see below? How do you know?

```
5  class Point {
6   public:
7    Point();
8    Point(const int x, const int y);
9    Point(const Point &copyme);
10   Point(const int* vec_array);
11
```

CSE 374: Lecture 24

Inheritance

Subclasses



- **Polymorphism.** In essence, polymorphism is the ability access different objects through the same *interface*. For instance, if you have an interface that represents an electronic device, that interface would have the ability to turn the device on and off. You can use the actual physical types - computer, phone, television, etc - as if they were an electronic device, because they all have the on/off capability.
- **Inheritance.** This is one of the meatiest pieces of OO programming. Inheritance allows the sharing of BEHAVIORS. For instance, a Square is a type of Rectangle, and has the same way to compute its area (width times height) - therefore by make Square inherit from Rectangle, we can share that behavior and avoid duplicating the code.

Subclassing supports fundamental Object Oriented precepts. It may not always be what we want, but understanding OO requires understanding subclassing.

Motivation

and C++ Syntax

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

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

Inheritance

A parent-child “is-a” relationship between classes

- A child (derived class) extends a parent (base class)

Terminology:

Java	C++
Superclass	Base Class
Subclass	Derived Class

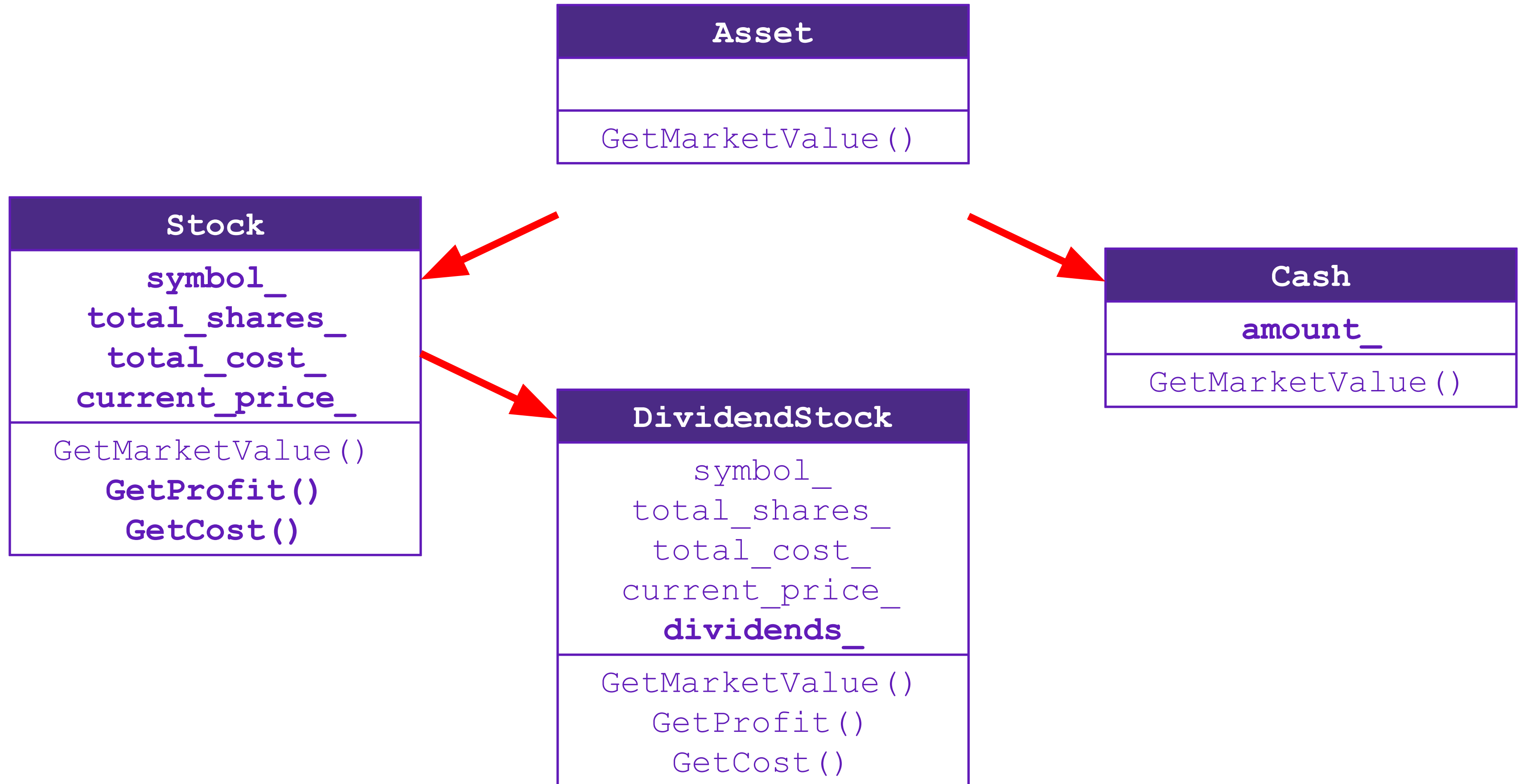
- Java: Subclass inherits from super class. (Superclass is “higher” in the hierarchy)
- C++: Derived class inherits from base class. (Base class is “higher” in the hierarchy)
    - Think of derived class as a derivative of the base class (e.g. car class is a derivative of vehicle class)
    - Mean the same things. You’ll hear both.

Inheritance

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

- `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
- Derived classes must have access but clients should not be allowed

`protected` isn't truly protected as an adversary client can just extend a protected class and get access to the "protected" information. Hence its use is rather limited in the real world.

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

Back to Stocks

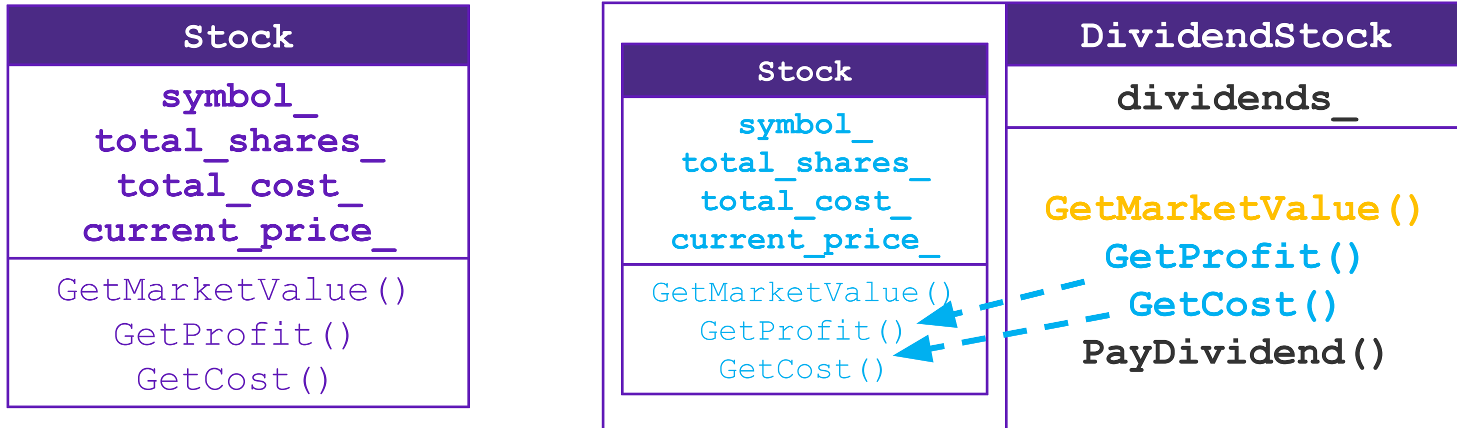
Stock
symbol_
total_shares_
total_cost_
current_price_
GetMarketValue()
GetProfit()
GetCost()

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

BASE

DERIVED

Back to Stocks



A derived class:

- **Inherits** the behavior and state (specification) of the base class
- **Overrides** some of the base class' member functions (optional)
- **Extends** the base class with new member functions, variables (optional)

Constructing and Destructing

- Constructor of base class gets called before constructor of derived class
  - Default (zero-arg) constructor unless you specify a different one after the : in the constructor
  - Initializer syntax: `Foo::Foo(…): Bar(args); it(x) { … }`
  - Needed to execute base class constructor with arguments; also works on instance variables and is preferred in production code (slogan: “initialization preferred over assignment”)
- Destructor of base class gets called after destructor of derived class
- So constructors/destructors really extend rather than override
  - Typically what you want
  - Java is the same

Mystery.cpp

Constructing and Destructing

- Constructor of base class gets called before constructor of derived class
  - Default (zero-arg) constructor unless you specify a different one after the : in the constructor
  - Initializer syntax: `Foo::Foo(…): Bar(args); it(x) { … }`
  - Needed to execute baseclass constructor with arguments; also works on instance variables and is preferred in production code (slogan: "initialization preferred over assignment")
- Destructor of base class gets called after destructor of derived class
- So constructors/destructors really extend rather than override
  - Typically what you want
  - Java is the same

```
class Derived: public Base {
public:
    double m_cost;
    Derived(double cost=0.0, int id=0)
        : Base{ id }, // Call Base(int) constructor
          m_cost{ cost }  // assign parameter values
    {
    }
    double getCost() const { return m_cost; }
};
```

Mystery.cpp

Method Override

If a derived class defines a method with the same method name and argument types as one defined in the base class, it *overrides* (i.e., replaces) the base class method.

Remember constructors EXTEND, new methods OVERRIDE

If you want to use the base-class code, you specify the base class when making a method call (`base::method(…)`)

Like super in Java (no such keyword in C++ since there may be multiple inheritance)

Questions?

Polymorphism

& Dynamic Dispatch

Polymorphism 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`
- `PromisedType` defines the *interface* (*i.e.* what can be called on `var_p`), but `ActualType` may determine which *version* gets invoked

Analogy: A box labeled “cell phone” could hold Android or iPhone

- `PromisedType` is the box, `ActualType` is the Android or iPhone

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
- This is the behavior in Java

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

Is this a Stock or a DividendStock ?

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

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

Requesting Dynamic Dispatch

Prefix the member function declaration with the `virtual` keyword

- Derived/child functions don't need to repeat `virtual`, but is 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
- C++ doesn't do dynamic dispatch by default so virtual keyword is strictly required if we want to make sure we're calling the most derived version of a function.

`override` keyword (C++11)

- Tells compiler this method should be overriding an inherited virtual function – **always** use when you can
- Prevents overloading vs. overriding bugs

Static vs Dynamic Types

- Suppose we have a variable declared
  `T* x`
  and a method call
  `x->f(`*params*`)`
- There are *two types* associated with `x`:
  - Static type: the declared type of `x`, which is `T` here
  - Dynamic type: the actual type of the object `*x`, which will either be `T` *or some subclass (subtype) of* `T`
    - And this *can change during execution* if `x` is changed to point to different objects with different (sub)types of `T`

Obtaining Dynamic Dispatch

- Static type (compile-type type) must differ from the dynamic type (actual runtime type of the object)
  - Therefore, need to have some form of indirection (eg, a pointer or reference)
- The member function in the static type must be declared `virtual`

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

DividendStock dividend;
DividendStock* dp = &dividend;
Stock stock;
Stock* sp = &dividend;

dp->GetMarketValue();
sp->GetMarketValue();
stock.GetMarketValue();
```

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 {   // inherited
  return GetMarketValue() - GetCost();
}
```

Should call 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

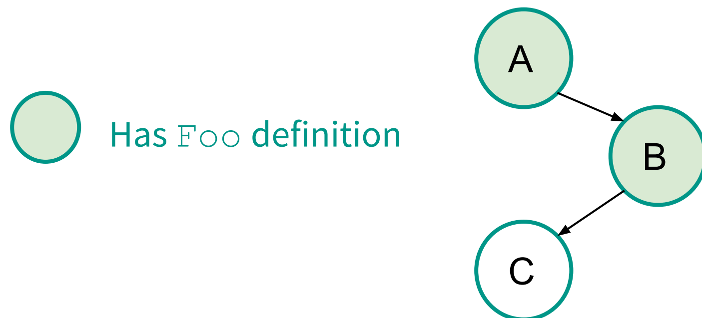
```
#include "Stock.h"
#include "DividendStock.h"

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();
```

A DividendStock “is-a” Stock, and has every part of Stock’s interface

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()
};
```

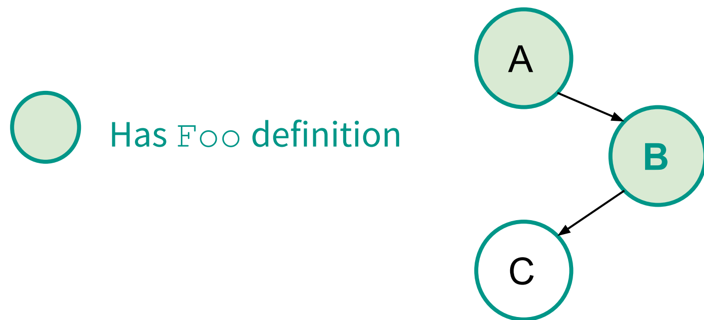
```
void Bar() {
  A* a_ptr;
  C c;

  a_ptr = &c;

  // Whose Foo() is called?
  a_ptr->Foo();
}
```



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;

  // B::Foo() is called
  a_ptr->Foo();
}
```



Questions?

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

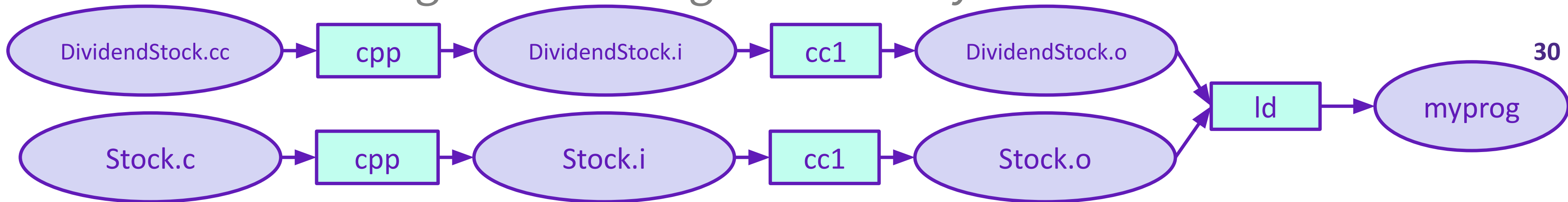
```
virtual double Stock::GetMarketValue() const;
virtual double Stock::GetProfit() const;
```

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

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

Stock.cc

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`**`()` vs. `DividendStock::`**`GetMarketValue`**`()` vs. something else that might not exist yet?



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`**`()`
    vs. `DividendStock::`**`GetMarketValue`**`()`
    vs. something else that might not exist yet?
  - ***Function pointers!***

Virtual Table

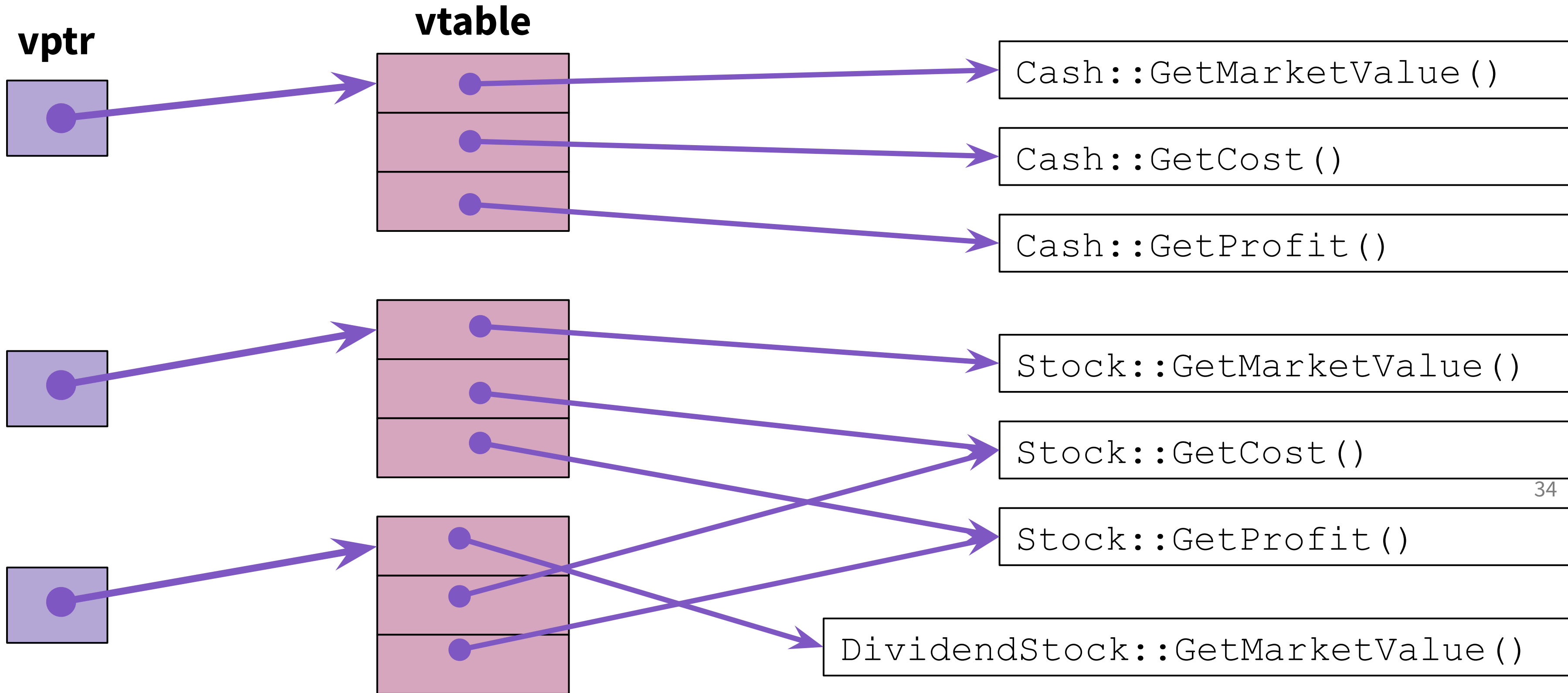
Virtual tables & virtual table pointers

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 newly constructed object’s class
  - Thus, the vptr “remembers” what class the object is

vptr and vtable Visualization



vtable/vptr Example

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

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

class Der2 : public Base {
 public:
  virtual void f2();
};
```

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

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

b0ptr->f1();
b0ptr->f2();

b1ptr->f1();
b1ptr->f2();

d2.f1();
b2ptr->f1();
b2ptr->f2();
```

vtable/vptr Example

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

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

class Der2 : public Base {
 public:
  virtual void f2();
};
```

```
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()

d2.f1();        Base::f1()
b2ptr->f1();    Base::f1()
b2ptr->f2();    Der2::f2()
```

Static Dispatch

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 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 0;
}
```

```
Derived::foo()
...
```

```
Base::foo()
...
```

Static Dispatch Example

Removed `virtual` on methods:

Stock.h

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

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

ds->GetMarketValue(); // Calls DividendStock::GetMarketValue()
s->GetMarketValue();  // Calls Stock::GetMarketValue()

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

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

`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

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 zero 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++, you can pick what you want

- Omitting `virtual` can cause obscure bugs
- (Most of the time, you want member function to be virtual)

Abstract Classes

Sometimes we want to include a function in a class but *only* implement it in derived classes

- In Java, we would use an abstract method
- In C++, we use a "pure virtual" function
  - Example: `virtual string noise() = 0;`

A class containing *any* pure virtual methods is abstract

- You can't create instances of an abstract class
- Extend abstract classes and override methods to use them

A class containing *only* pure virtual methods is the same as a Java interface

- Pure type specification without implementations (e.g. `asset`)

Pure virtual methods and interfaces (?)

- A C++ “pure virtual” method is like a Java “abstract” method.
  - Subclass must override because there is no definition in base class
- Makes sense with dynamic dispatch
- Funny syntax in base class; override as usual:

```
class C {
    virtual t0 m(t1,t2,...,tn) = 0;
    ...
};
```

- Side-comment: with multiple inheritance and pure-virtual methods, no need for a separate notion of Java-style interfaces

Questions?

(Up) casting

- An object of a derived class *cannot* be cast to an object of a base class.
  - For the same reason a `struct T1 {int x,y,z;}` cannot be cast to type `struct T2 {int x,y;}` (different size)
- A pointer to an object of a derived class *can* be cast to a pointer to an object of a base class.
  - For the same reason a `struct T1*` can be cast to type `struct T2*` (pointers to a location in memory)
  - (Story not so simple with multiple inheritance)
- After such an *upcast*, field-access works fine (prefix)
  - but what do method calls mean in the presence of overriding? (see virtual)

(Down) casting

- C pointer-casts: unchecked; be careful
- Java: checked; may raise ClassCastException
- New: C++ has "all the above" (several different kinds of casts)
    - If you use single-inheritance and know what you are doing, the C-style casts (same pointer, assume more about what is pointed to) should work fine for downcasts
    - Worth learning about the differences on your own