C++ STL Continued, Inheritance

CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Audrey Seo

Deeksha Vatwani

Derek de Leuw

Katie Gilchrist

Administrivia

- ex11 (STL Vector) due *Saturday* (tomorrow) night, 11 pm
  - Unusual deadline because of hw2 yesterday and midterm Monday
- New ex12 (STL map) out today, due Wed. 10 am (usual time)
- HW3 writeup on web now. Starter code will be pushed this weekend & demo in class today
  - Get started *immediately* after Monday's midterm – don't wait

Administrivia

- Midterm Monday, in-class
  - Everything up through core C++ but not templates/STL, inheritance
  - Can bring one hand-written notecard for reference during the exam (blank cards available after class)
  - Review session Sun. 1pm, MGH 241. ~1 hour or a bit longer if needed. Bring your questions!

Lecture Outline

- **More STL Containers**
- C++ Inheritance

Reference: More info in *C++ Primer* §9.2, 11.2

STL Containers ☺ (review)

- A container is an object that stores (in memory) a collection of other objects (elements)
  - Implemented as class templates, so hugely flexible

- Several different classes of container
  - Sequence containers (`vector`, `deque`, `list`, `...`)
  - Associative containers (`set`, `map`, `multiset`, `multimap`, `bitset`, `...`)
  - Differ in algorithmic cost and supported operations

STL `iterator` (review)

- Each container class has an associated iterator class (*e.g.* `vector<int>::iterator`) used to iterate through elements of the container
  - http://www.cplusplus.com/reference/std/iterator/
  - Iterator range is from `.begin()` up to `.end()`
    - `end` is one past the last container element!
  - Some container iterators support more operations than others
    - All can be incremented (`++`), copied, copy-constructed
    - Some can be dereferenced on RHS (*e.g.* `x = *it;`)
    - Some can be dereferenced on LHS (*e.g.* `*it = x;`)
    - Some can be decremented (`--`)
    - Some support more (`[]`, `+`, `-`, `+=`, `-=`, `<`, `>` operators)

STL Algorithms (review)

- A set of functions to be used on ranges of elements
  - Range: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
  - General form: `algorithm(begin, end, ...);`
- Algorithms operate directly on range *elements* rather than the containers they live in
  - Make use of elements' copy constructor, =, ==, !=, <
  - Some do not modify
    - *e.g.* find, count, for_each, min_element, binary_search
  - Some do modify
    - *e.g.* sort, transform, copy, swap

More STL Containers

See:

https://courses.cs.washington.edu/courses/cse333/25su/lecture/_14-c++-STL-cont+inheritance-example/

Unordered Containers (C++11)

- `unordered_map`, `unordered_set`
  - And related classes `unordered_multimap`, `unordered_multiset`
  - Average case for key access is O(1)
    - But range iterators can be less efficient than ordered `map`/`set`
  - See *C++ Primer*, online references for details

Lecture Outline

- More STL Containers
- **C++ Inheritance**
  - Review of basic idea
  - Dynamic Dispatch
  - vtables and vptr

- Reference: *C++ Primer*, Chapter 15

Overview of Next Two Inheritance 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 (i.e., dynamic dispatch) are optional
    - Pure virtual functions, abstract classes, why no Java "interfaces"
    - Assignment slicing, using class hierarchies with STL

Overview of Next Two Inheritance 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 (i.e., dynamic dispatch) are optional
    - *Pure virtual functions, abstract classes, why no Java "interfaces"*
    - *Assignment slicing, using class hierarchies with STL*

Lecture Outline

- More STL Containers
- **C++ Inheritance**
  - **Review of basic idea**
  - Dynamic Dispatch
  - vtables and vptr

- Reference: *C++ Primer*, Chapter 15

Stock Portfolio Example

- A portfolio that 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

- See sample code in `initial_design/`

Inheritance

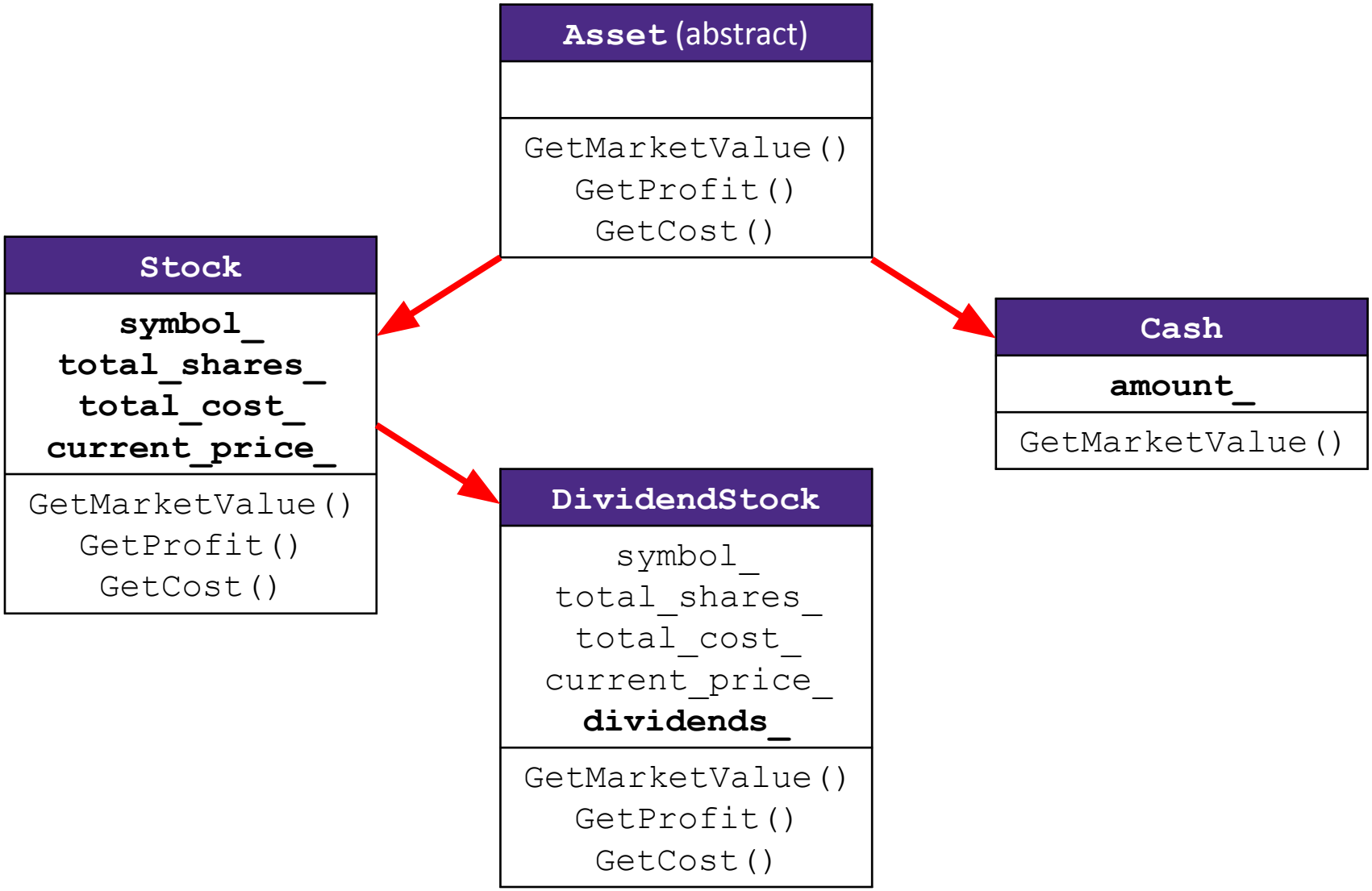
- A parent-child "is-a" relationship between classes
  - A child (derived class) extends a parent (base class)
- Benefits:
  - Code reuse
    - Children can automatically inherit code from parents
  - Polymorphism
    - Ability to redefine existing behavior but preserve the interface
    - Children can 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

Terminology

Java	C++
Superclass	Base Class
Subclass	Derived Class

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

Design With Inheritance



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
  - (recall that C++ style guide says all *data members* should be private; your getters/setters must, minimally, be `protected`)

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

```
class Name : public BaseClass, public OtherBaseClass {
```

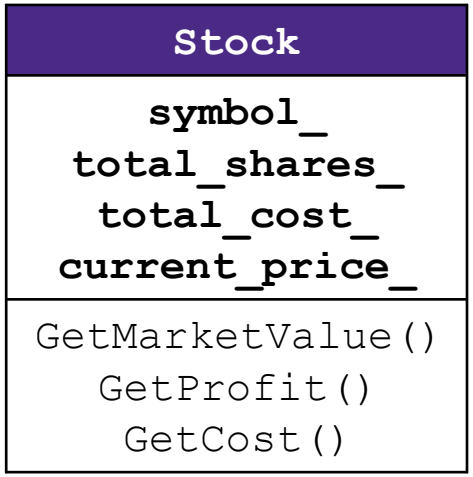
Class derivation List

- Comma-separated list of classes to inherit from

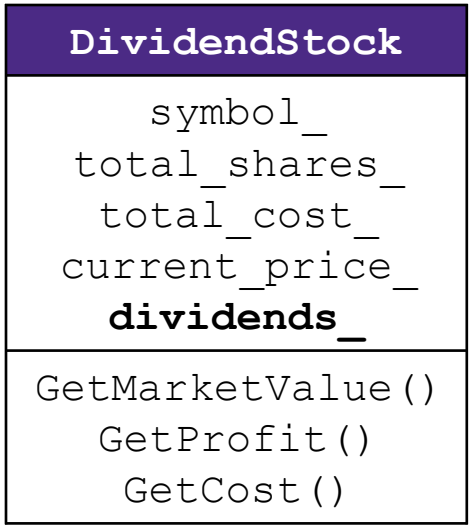
```
#include "BaseClass.h"

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

- 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 (in spite of sloppy description in some books that say otherwise)

Back to Stocks

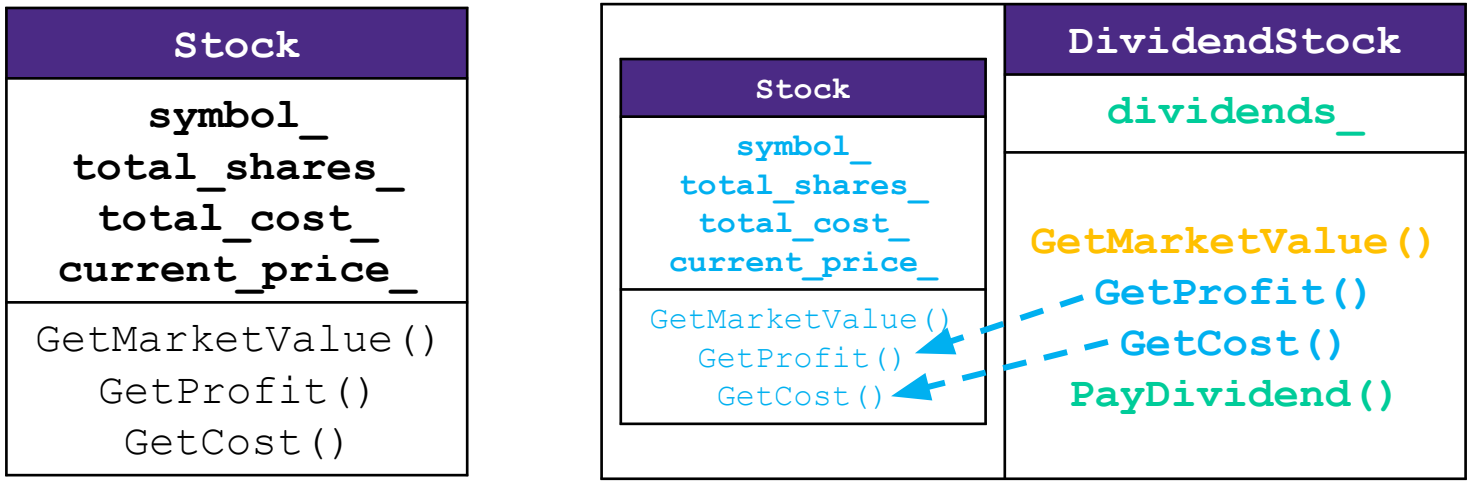


BASE



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 (opt.)
  - **Extends** the base class with new member functions, variables (opt.)

Lecture Outline

- More STL Containers
- **C++ Inheritance**
  - Review of basic idea
  - **Dynamic Dispatch**
  - vtables and vptr

- Reference: *C++ Primer*, Chapter 15

Like Java: Dynamic Dispatch

- 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 similar to 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

Like Java: Dynamic Dispatch

- 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: PrintStock(Stock *s) { s->Print() }
  - Calls Print() function appropriate to Stock, DividendStock, etc. without knowing the exact class of *s, other than it is some sort of Stock
  - So the Stock (DividendStock, etc.) object *itself* has to carry some sort of information that can be used to decide which Print() to call
  - (see inherit-design/useasssets.cc)

Requesting Dynamic Dispatch

- Prefix the member function declaration with the `virtual` keyword
  - Derived functions don't need to repeat `virtual`, since it's virtual in all subclasses, 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

Requesting Dynamic Dispatch

- `virtual` keyword
- `override` keyword (C++11)
- Both of these are technically *optional* in derived classes
  - A virtual function is virtual in all subclasses as well
  - A function with the same signature as a virtual function in the superclass always overrides
  - Be consistent and follow local conventions

Dynamic Dispatch Example

```
class Stock {
...
    virtual double GetMarketValue() const;
    virtual double GetProfit() const;
...
}
```

Stock.h

```
class DividendStock, public Stock {
...
    override double GetMarketValue() const;
...
}
```

DividendStock.h

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

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

Stock.cc

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

double DividendStock::GetProfit() const {  // inherited
  return GetMarketValue() - GetCost();
}       // really Stock::GetProfit()
```

DividendStock.cc

Dynamic Dispatch Example

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

DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;
// Invokes DividendStock::GetMarketValue()
ds->GetMarketValue();

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

Dynamic Dispatch Example

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

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

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

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

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();
}
```

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 

Lecture Outline

- More STL Containers
- **C++ Inheritance**
  - Review of basic idea
  - Dynamic Dispatch
  - **vtables and vptr**

- Reference: *C++ Primer*, Chapter 15

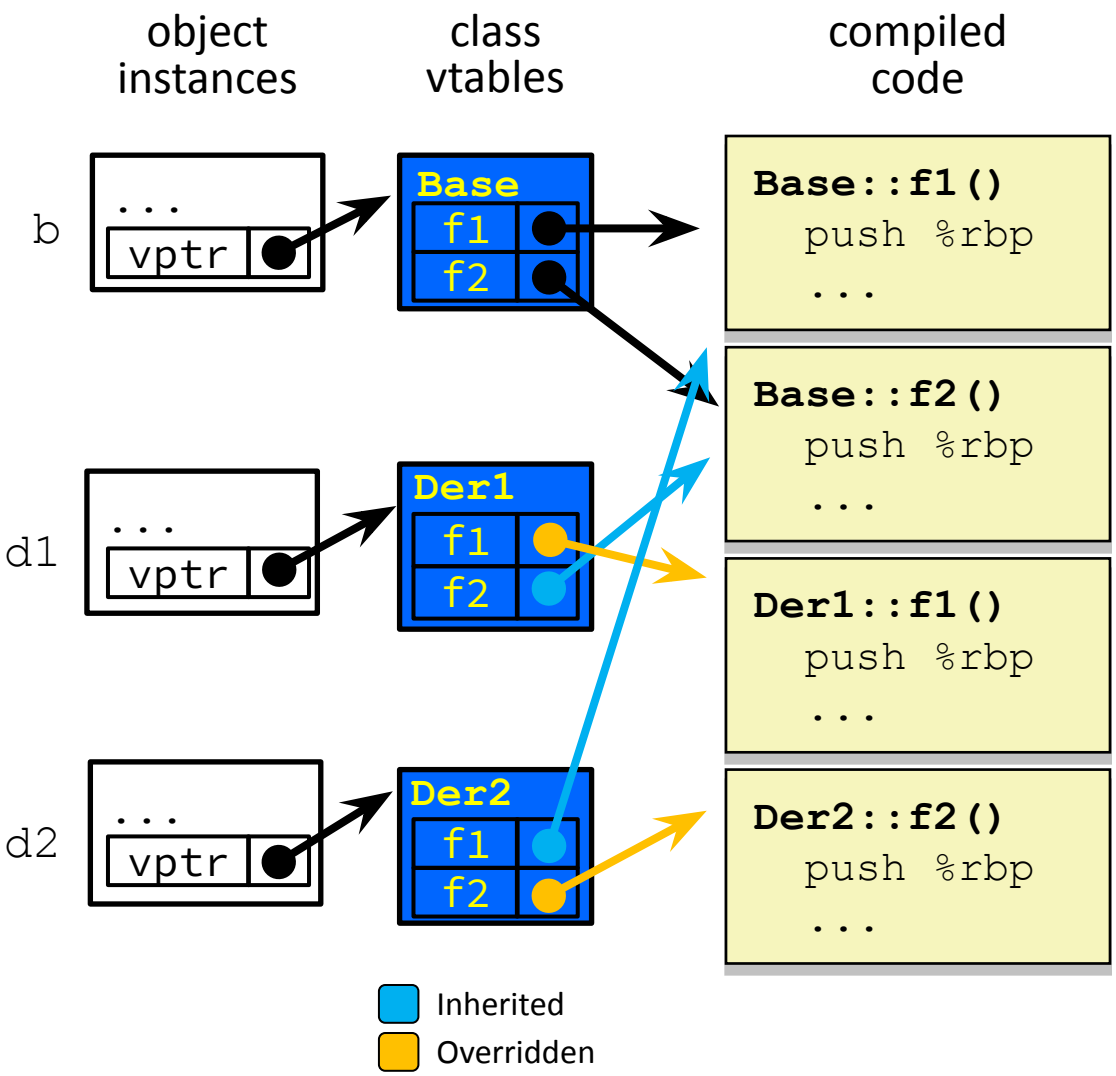
vtables and the vptr

- 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) in *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

vtable/vptr Example

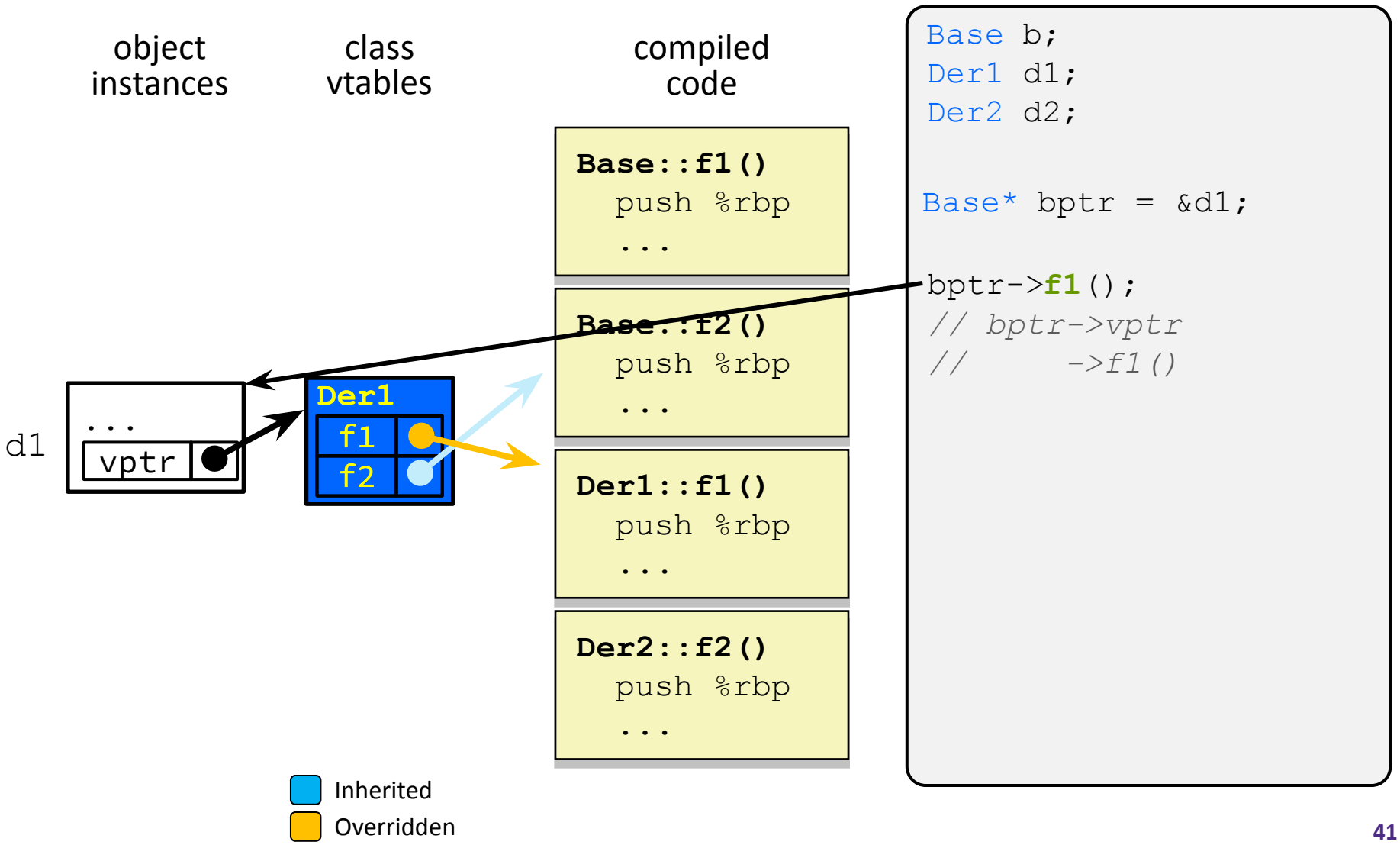
```
class Base {
 public:
  virtual void f1();
  virtual void f2();
};

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

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

vtable/vptr Example

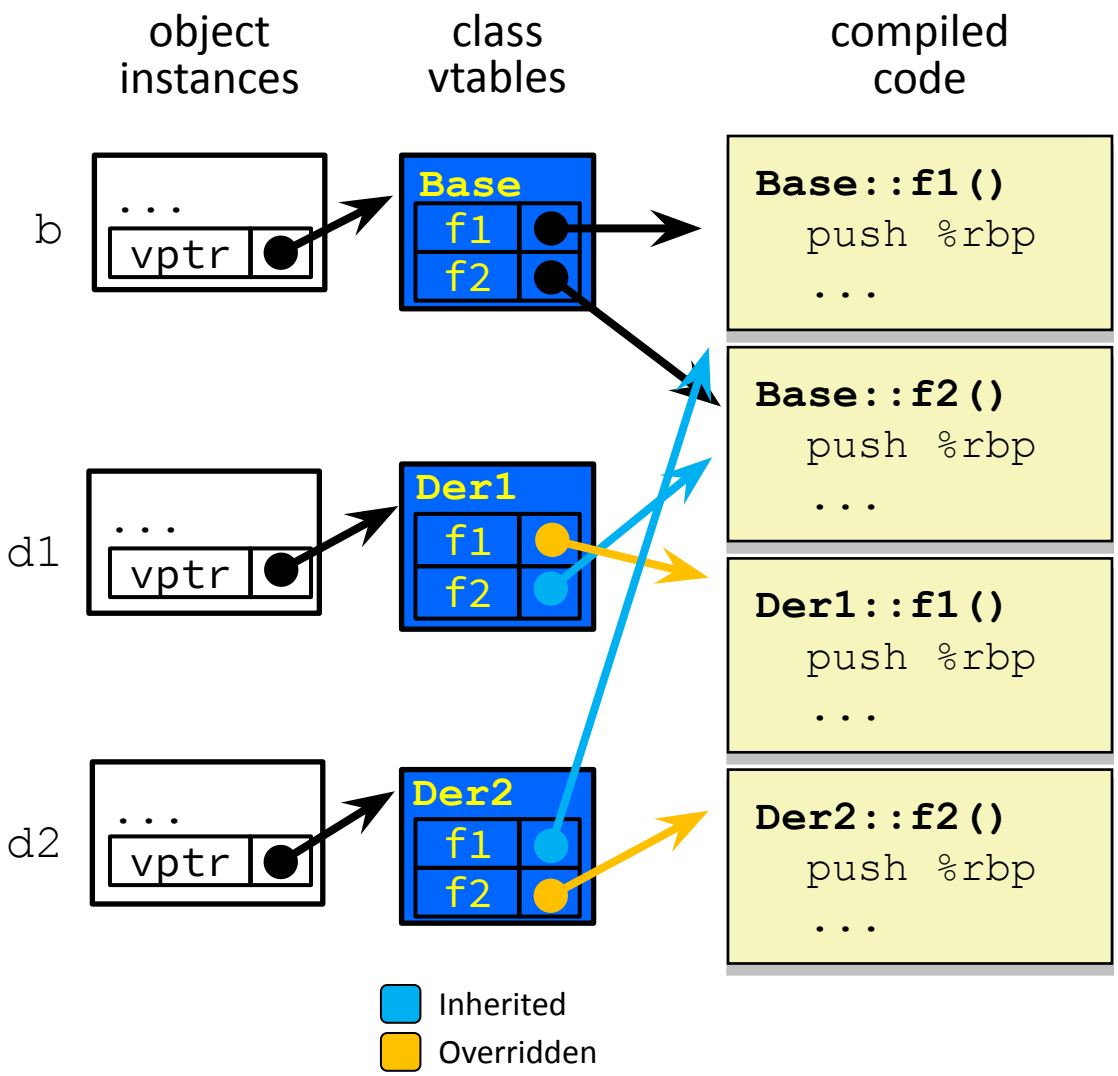



```
Base b;
Der1 d1;
Der2 d2;

Base* bptr = &d1;

bptr->f1();
```

vtable/vptr Example

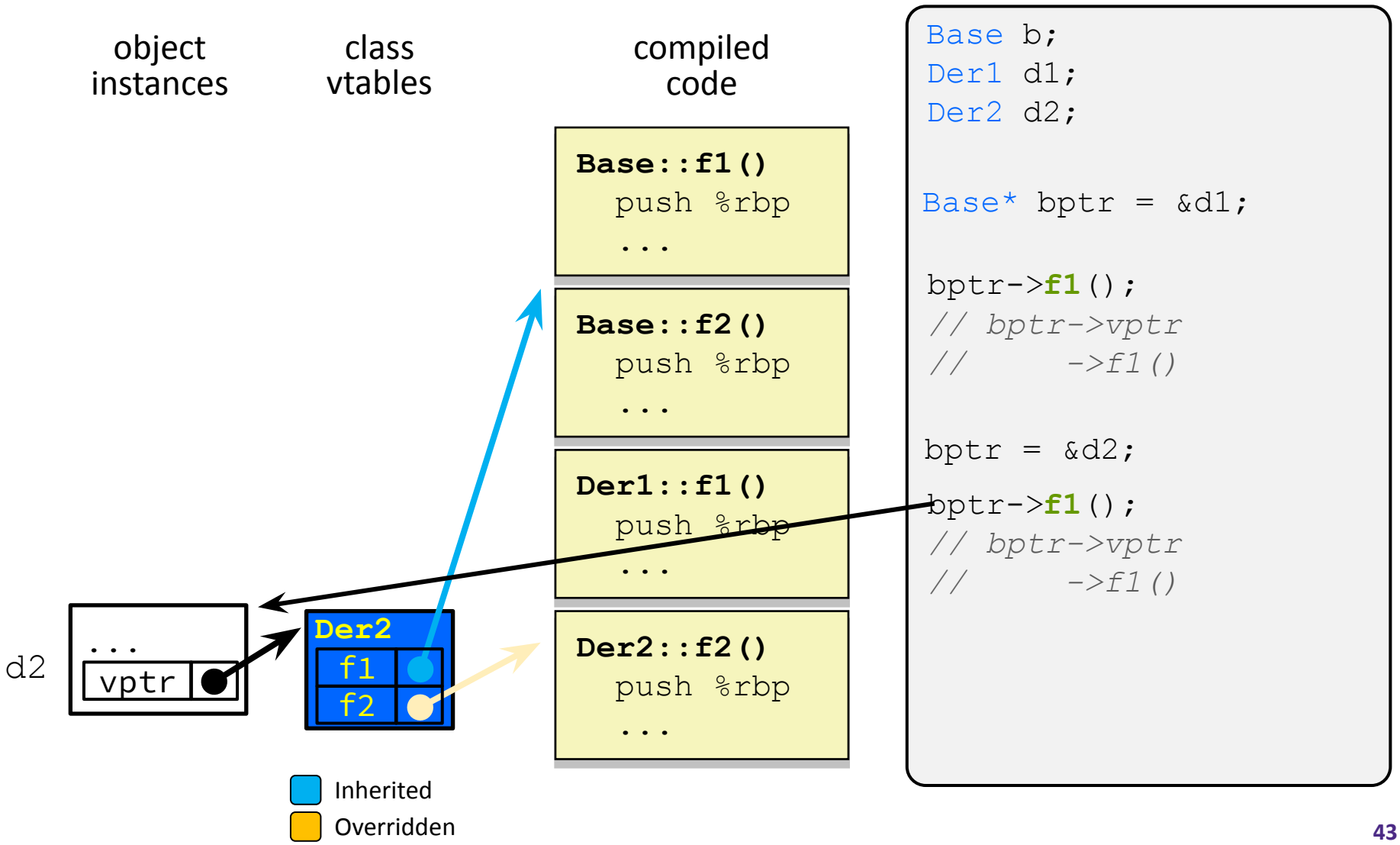


vtable/vptr Example




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

Base* bptr = &d1;

bptr->f1();
// bptr->vptr
//      ->f1()

bptr = &d2;

bptr->f1();
// bptr->vptr
//      ->f1()
```

vtable/vptr Example



Don't forget!

- Midterm Monday, in-class
- Ex 11 due on Saturday
- Ex 12 due on Monday
- HW3 write-up released, repos pushed soon

Extra Exercise #1

- Take one of the books from HW2's `test_tree` and:
  - Read in the book, split it into words (you can use your hw2)
  - For each word, insert the word into an STL `map`
    - The key is the word, the value is an integer
    - The value should keep track of how many times you've seen the word, so each time you encounter the word, increment its map element
    - Thus, build a histogram of word count
  - Print out the histogram in order, sorted by word count
  - Bonus: Plot the histogram on a log-log scale (use Excel, gnuplot, etc.)
    - x-axis: log(word number), y-axis: log(word count)