# C++ Inheritance I
## CSE 333

**Instructor:**      Hannah C. Tang

**Teaching Assistants:**

| | | |
|---|---|---|
| Deeksha Vatwani | Hannah Jiang | Jen Xu |
| Leanna Nguyen | Nam Nguyen | Sayuj Shahi |
| Tanay Vakharia | Wei Wu | Yiqing Wang |
| Zohar Le | | |

# HW3 Tips

❖ HW3 writes some pretty big index files

  ▪ Hundreds of thousands of write operations - no problem for today's fast machines and disks!!

❖ Except…

  ▪ If you're running on attu or a CSE lab linux workstation, every write to your personal directories goes to a network file server(!)

   • ∴ Lots of slow network packets vs full-speed disks — can take much longer to write an index to a server vs. a few sec. locally (!!)

   • Suggestion: write index files to `/tmp/...`, which is a local disk and very fast.  But please clean up when you're done.

# HW3 Tips

❖ Do your main debugging on a tiny set of files in a just a few nested directories

  ▪ Goal: dataset is tiny enough that you can draw pictures of what should be happening

  ▪ Because you can manually verify in gdb and with disk file tools!

❖ See section materials about visualizing disk data!

# Lecture Outline

- ❖ **C++ Inheritance**
  - ■ **Review of basic idea**
  - ■ **Dynamic Dispatch**
  - ■ **vtables and vptr**

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

# 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 (i.e., dynamic dispatch) are optional

  - Pure virtual functions, abstract classes, why no Java "interfaces"

  - Assignment slicing, using class hierarchies with STL

❖ Casts in C++

❖ Reference: C++ Primer, ch. 15

# 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 (i.e., dynamic dispatch) are optional

  • *Pure virtual functions, abstract classes, why no Java "interfaces"*

  • *Assignment slicing, using class hierarchies with STL*

❖ *Casts in C++*


❖ Reference: C++ Primer, ch. 15

- (read it! a lot of how C++ does this looks like Java, but details differ)

# 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_<br>total_shares_<br>total_cost_<br>current_price_ |
| GetMarketValue()<br>GetProfit()<br>GetCost() |

| DividendStock |
| --- |
| symbol_<br>total_shares_<br>total_cost_<br>current_price_<br>dividends_ |
| GetMarketValue()<br>GetProfit()<br>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
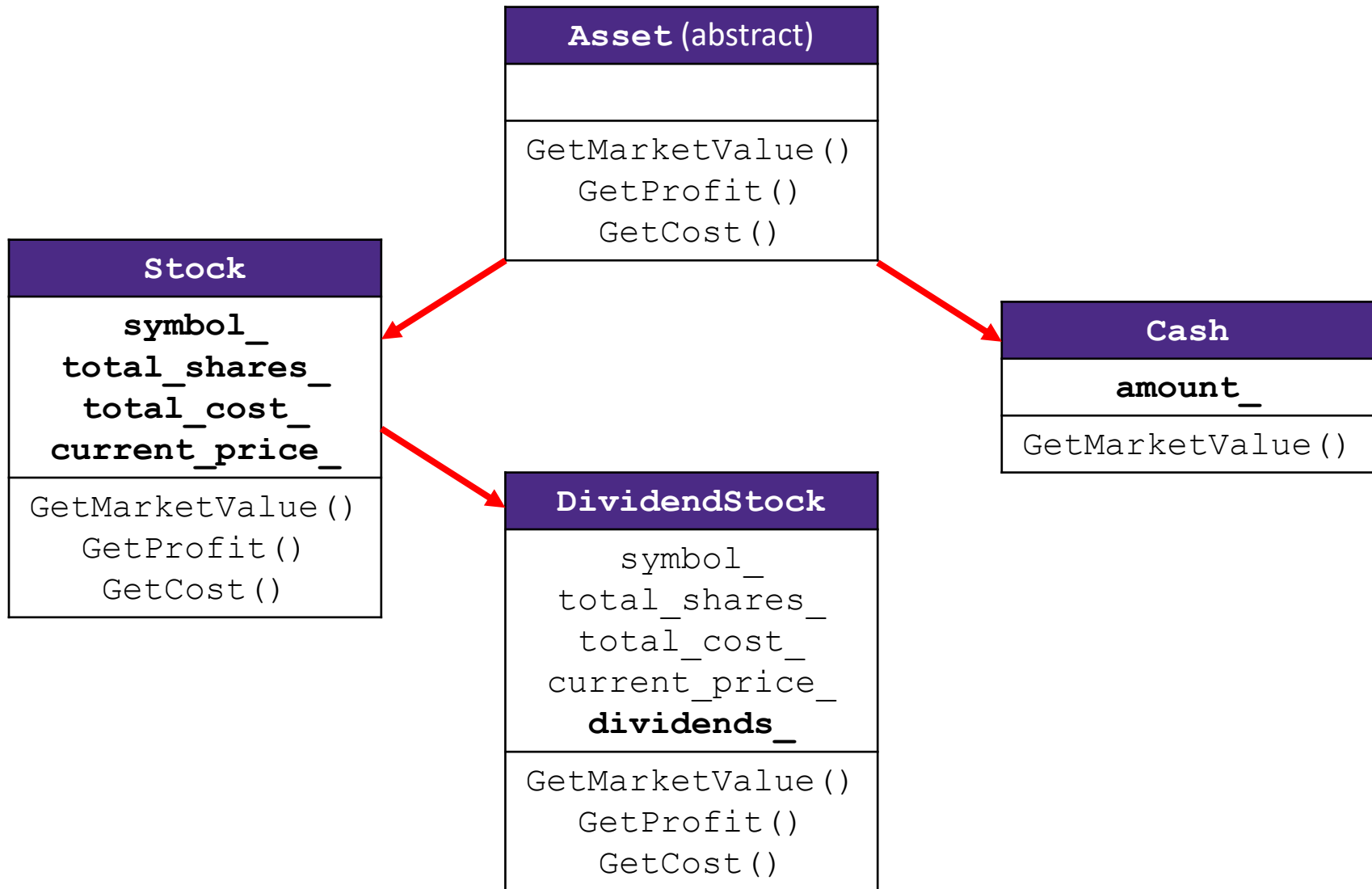
# Like Java: Access Modifiers

- ❖ `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


❖ 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
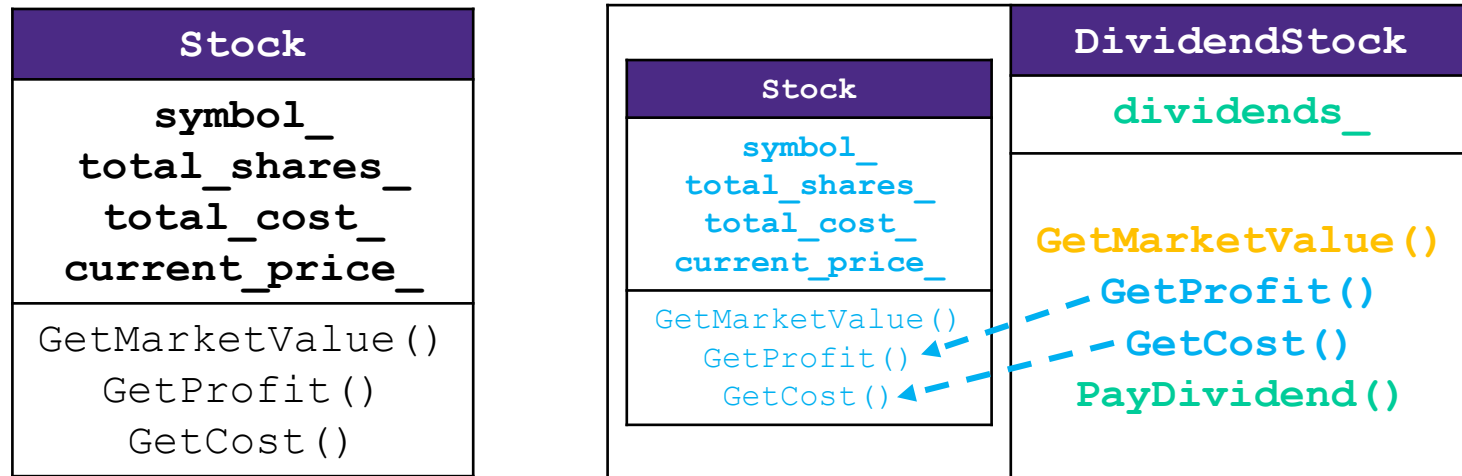
# Back to Stocks

| Stock |
|---|
| **symbol_**<br>**total_shares_**<br>**total_cost_**<br>**current_price_** |
| GetMarketValue()<br>GetProfit()<br>GetCost() |

| DividendStock |
|---|
| symbol_<br>total_shares_<br>total_cost_<br>current_price_<br>**dividends_** |
| GetMarketValue()<br>GetProfit()<br>GetCost() |

BASE                    DERIVED

# Back to Stocks

| Stock |
|---|
| **symbol_** <br> **total_shares_** <br> **total_cost_** <br> **current_price_** |
| GetMarketValue() <br> GetProfit() <br> GetCost() |

| Stock | DividendStock |
|---|---|
| **symbol_** <br> **total_shares_** <br> **total_cost_** <br> **current_price_** | **dividends_** |
| GetMarketValue() <br> GetProfit() <br> GetCost() | **GetMarketValue()** <br> **GetProfit()** <br> **GetCost()** <br> **PayDividend()** |

❖ 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.)

# 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

❖ 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

❖ Both of these are technically *optional* in derived classes

- A virtual function is virtual in all subclasses as well
- Be consistent and follow local conventions

# 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 <u>run time</u> 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();
}        // really Stock::GetProfit()
```
DividendStock.cc

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


double Stock::GetProfit() const {
  return GetMarketValue() – GetCost();
}
```
Stock.cc

# Poll Everywhere

❖ **Which version of** `GetMarketValue()` **is called?**

A
C

```cpp
int main(void)
  DividendStock *ds = ...
  cout << ds->GetProfit() << endl;
}
```

S
DS

*Engineering Days*

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

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

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

# Dynamic Dispatch Example

```cpp
#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();
```

# Most-Derived

```cpp
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()
};
```

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

  a_ptr = &c;

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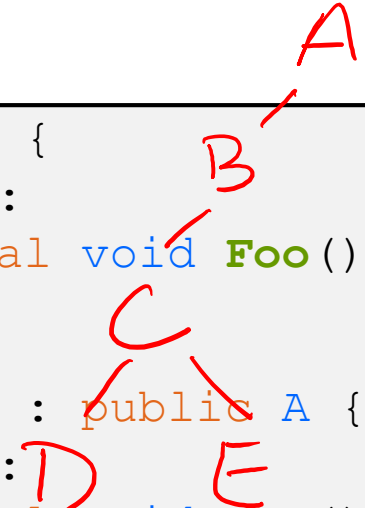
**Poll Everywhere**

❖ Whose **Foo**() is called?

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

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

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

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

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

class C : public B {
};

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

class E : public C {
};
```

|     | Q1 | Q2 |
|-----|----|----|
| A.  | A  | B  |
| B.  | A  | D  |
| C.  | B  | B  |
| D.  | B  | D  |
| E.  | We're lost… | |

**Poll Everywhere**

- ❖ Consider a hash table that maps strings to function pointers
  - Could you use this to selectively declare how the HW1 HashTable and LinkedLists should clean up its members?
  - What would happen if you could change the hash table mid-program?

```cpp
int main() {
  HashTable ht;

  // … initialization …

  *( ht.Find("Say Hi!!") )("你好");
  ht.Insert("Say Hi!!", printf);
  *( ht.Find("Say Hi!!") )("مرحبًا ");

  return EXIT_SUCCESS;
}
```

# Administrivia

- ❖ New exercise today – C++ inheritance mechanics: define an abstract class and two subclassses.

- ❖ HW3 due in a 1.5 weeks

- ❖ On track to return HW1 before HW3 deadline;  🤞 for HW2 as well!

# Lecture Outline

- ❖ **C++ Inheritance**
    - ▪ **Review of basic idea**
    - ▪ **Dynamic Dispatch**
    - ▪ **vtables and vptr**

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

# 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 newly constructed object's class
  - Thus, the vptr "remembers" what class the object is

# vtable/vptr Example

```cpp
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();
};
```

```cpp
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()
```
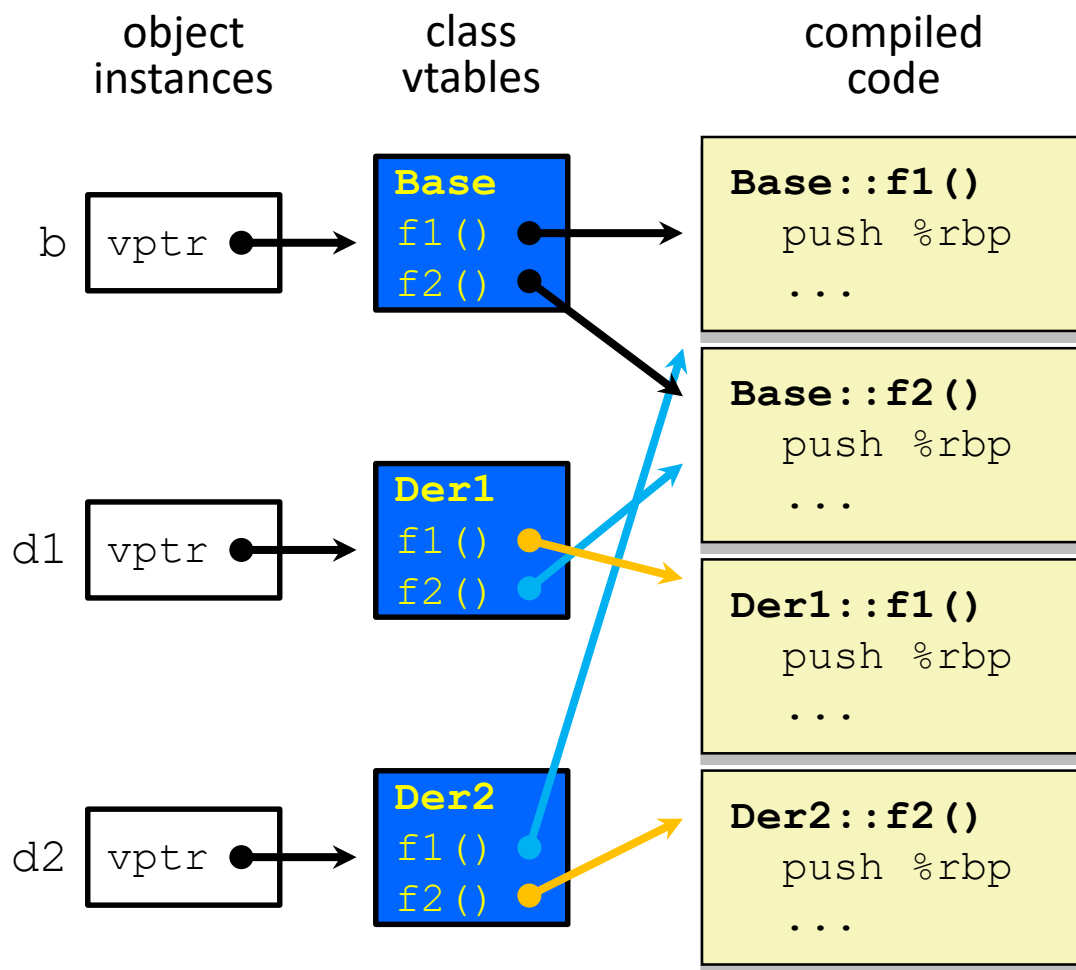
# vtable/vptr Example



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

Base* bptr = &d1;

bptr->f1();
// bptr -->
// d1.vptr -->
// Der1.vtable.f1 -->
// Der1::f1()

bptr = &d2;

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

# Let's Look at Some Actual Code

❖ Let's examine the following code using `objdump`

- `g++ -g -o vtable vtable.cc`
- `objdump -CDS vtable > vtable.d`

vtable.cc

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

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

int main(int argc, char** argv) {
  Der1 d1;
  d1.f1();
  Base* bptr = &d1;
  bptr->f1();
}
```

33

# More to Come…

Next time…