Poll Everywhere

pollev.com/cse333

Based on your midterm preparation and performance, what, if anything, do you plan to do differently to approach the final?

- Free response

Systems Programming

C++ Inheritance II

Instructor: Justin Hsia

Teaching Assistants:

Aadi Adgaonkar	Advaith Vijayakumar	Alice Fu
Andrew Vlahovich	Aviel Wood	Camden Harris
Esha Gore	Hannah Hempstead	Helena Wang
Laura Wang	Megan Frisella	Nina Tharamal
Shrihun Sankepally	Shriya Sateesh	Yusong Yan

Relevant Course Information

- Midterm regrade requests close at 11:59 PM tonight!
- Exercise 9 released today, due Wednesday
  - C++ smart pointers and inheritance: derived and abstract classes
- Homework 3 is due next Thursday (11/19)
  - Suggestion: Write index files to `/tmp/`, which is a local scratch disk and is very fast, but please clean up when you're done

Mid-Quarter Survey Feedback

- Lecture can feel fast or packed
  - Maybe more polls or worked examples; repeat student questions into mic
- Sections feel rushed, and TAs often can't finish
- Unpredictable wait times for the OH queue
  - Issues with check-in congestion and longer debugging sessions
- Exercise 4 was long and maybe underspecified
- Style grading can be a pain point, especially for C
- Drawing out Homework data structures was very useful
- Check-Ins are a mixed bag: most found them useful, but others found them unnecessary

Lecture Outline (1/3)

- **C++ Inheritance** (Reference: *C++ Primer*, Chapter 15)
  - **Abstract Classes**
  - **Static Dispatch**
  - Constructors and Destructors
  - Assignment
- C++ Casts

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 (*e.g.*, `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

Reminder: `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`

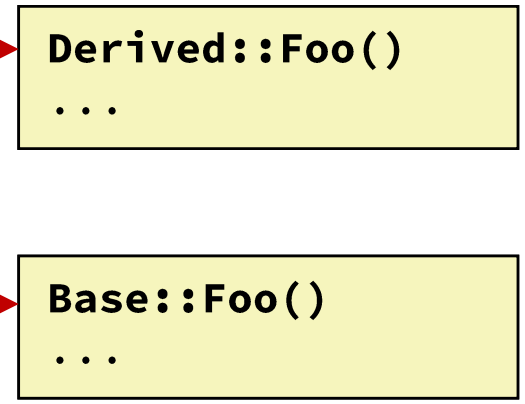
- `F()` will be called using dynamic dispatch even if overridden in a derived class without the `virtual` keyword
  - Good style to help the reader *and avoid bugs* by using `override`
    - Style guide controversy, if you use `override` should you use `virtual` in derived classes? Recent style guides 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:

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

Stock.h

defined in Stock & DividendStock

defined in Stock, inherited by DividendStock, 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();
```

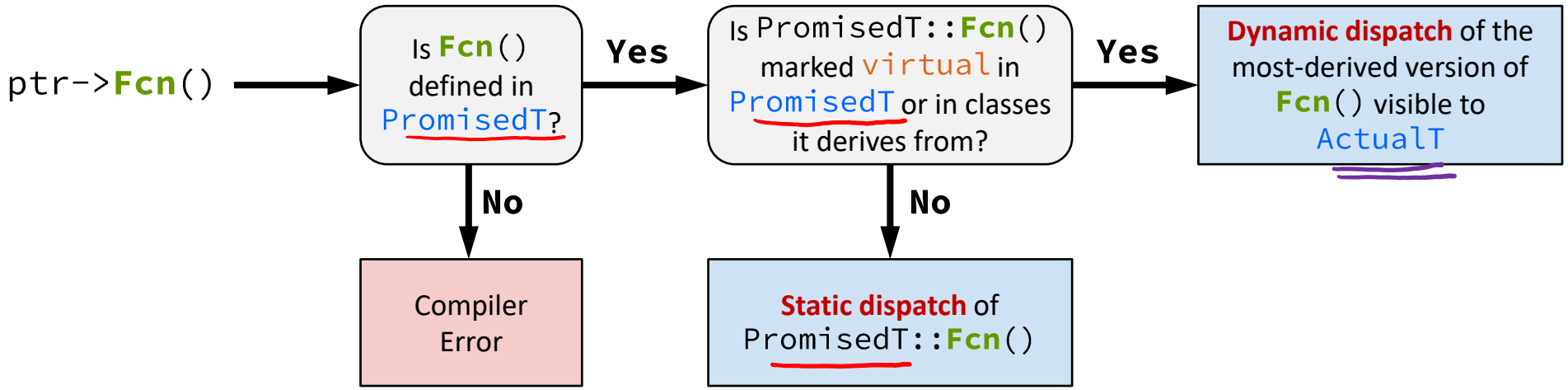
Why Not Always Use `virtual`?

- 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 derived class (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 (mostly want `virtual`)
  - Omitting `virtual` can cause obscure bugs

Mixed Dispatch

- Which function is called is a mix of (1) compile-time and runtime decisions, and (2) *how* you call the function
  - If called on an object (*e.g.*, `obj.Fcn()`), usually optimized into a hard-coded function call at compile time
  - If called via a pointer or reference (*e.g.*, `ptr->Fcn()`), use this flow chart:
    - Assume `PromisedT* ptr = new ActualT;`



Mixed Dispatch Example

```
class A {
 public:
  // m1 will use static dispatch
  void M1() { cout << "a1, "; }
  // m2 will use dynamic dispatch
  virtual void M2() { cout << "a2"; }
};

class B : public A {
 public:
  void M1() { cout << "b1, "; }
  // m2 is still virtual by default
  void M2() { cout << "b2"; }
};
```

(virtual)



static dispatch based on promised type
dynamic dispatch based on actual type

mixed.cc

```
void main(int argc,
          char* argv[]) {
  A a;
  B b;

  A* a_ptr_a = &a;
  A* a_ptr_b = &b;
  B* b_ptr_a = &a;  //Compiler error
  B* b_ptr_b = &b;

  a_ptr_a->M1();  // A::M1
  a_ptr_a->M2();  // A::M2

  a_ptr_b->M1();  // A::M1
  a_ptr_b->M2();  // B::M2

  b_ptr_b->M1();  // B::M1
  b_ptr_b->M2();  // B::M2
}
```

promised

actual

Poll Everywhere

pollev.com/cse333

Which versions of M1/M2 are invoked on a_ptr_b?

A. A::M1, A::M2

B. A::M1, B::M2

C. B::M1, A::M2

D. B::M1, A::M2

E. We're lost…

```
class A {
 public:
  void M1();
  virtual void M2();
};

class B : public A {
 public:
  void M1();
  void M2();
};

void main() {
  B b;
  A* a_ptr_b = &b;
  a_ptr_b->M1();
  a_ptr_b->M2();
}
```

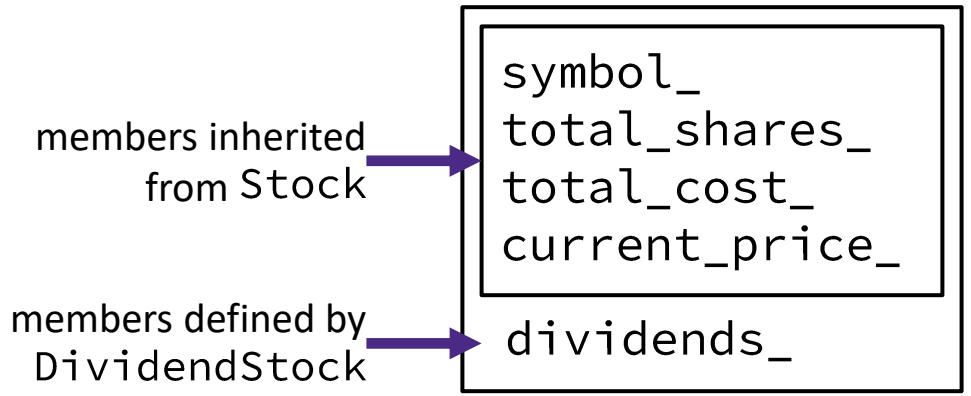
Lecture Outline (2/3)

- **C++ Inheritance** (Reference: *C++ Primer*, Chapter 15)
  - Abstract Classes
  - Static Dispatch
  - **Constructors and Destructors**
  - **Assignment**
- C++ Casts

Derived-Class Objects

- A derived object contains "subobjects" corresponding to the data members inherited from each base class
  - No guarantees about how these are laid out in memory (not even contiguousness between subobjects)

- Conceptual structure of `DividendStock` object:



Constructors and Inheritance

- A derived class **does not inherit** the base class' constructor
  - The derived class must have its own constructor
  - A synthesized default constructor for the derived class first invokes the default constructor of the base class and then initializes the derived class' member variables
    - Compiler error if the base class has no default constructor
  - The base class constructor is invoked *before* the constructor of the derived class
    - You can use the initialization list of the derived class to specify which base class constructor to use

Constructor Examples

badctor.cc

```
class Base {  // no default ctor
 public:
  Base(int yi) : y(yi) { }
  int y;
};

// Compiler error when you try to
// instantiate a Der1, as the
// synthesized default ctor needs
// to invoke Base's default ctor.
class Der1 : public Base {
 public:
  int z;
};

class Der2 : public Base {
 public:
  Der2(int yi, int zi)
    : Base(yi), z(zi) { }
  int z;
};
```

invokes a specific constructor

goodctor.cc

```
// has default ctor
class Base {
 public:
  int y;
};

// works now
class Der1 : public Base {
 public:
  int z;
};

// still works
class Der2 : public Base {
 public:
  Der2(int zi) : z(zi) { }
  int z;
};
```

Destructors and Inheritance



- Destructor of a derived class:
  - *First* runs body of the dtor
  - *Then* invokes of the dtor of the base class

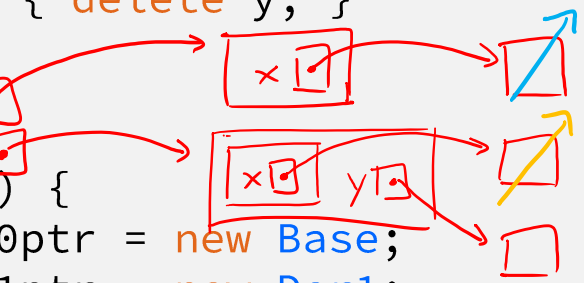
- Static dispatch of destructors is almost always a mistake!
  - Good habit to always define a dtor as virtual
    - Empty body if there's no work to do

```
class Base {
 public:
  Base() { x = new int; }
  ~Base() { delete x; }   // static dispatch
  int* x;
};

class Der1 : public Base {
 public:
  Der1() { y = new int; }
  ~Der1() { delete y; }
  int* y;
};

void Foo() {
  Base* b0ptr = new Base;
  Base* b1ptr = new Der1;

  delete b0ptr;  // deletes x
  delete b1ptr;  // only deletes x - leaks y!
}                // invokes ~Base()
```



baddtor.cc

Assignment and Inheritance

- C++ allows you to assign the value of a derived class to an instance of a base class
  - Known as object slicing
    - It's legal since b = d passes type checking rules
    - But b doesn't have space for any extra fields in d

```
class Base {
 public:
  Base(int xi) : x(xi) { }
  int x;
};

class Der1 : public Base {
 public:
  Der1(int yi) : Base(16), y(yi) { }
  int y;
};

void Foo() {
  Base b(1);
  Der1 d(2);

  d = b;   // compiler error – not enough info
  b = d;   // OK, but what happens to y?
}
```

b | x 1

d | x 16 | y 2

slicing.cc

- It can get weirder than this: https://stackoverflow.com/a/14461532

STL and Inheritance (1/2)

- Recall: STL containers store **copies of values**
  - What happens when we want to store mixes of object types in a single container? (*e.g.*, `Stock` and `DividendStock`)
  - You get sliced ☹

```
#include <list>
#include "Stock.h"
#include "DividendStock.h"

int main(int argc, char* argv[]) {
  Stock s;
  DividendStock ds;
  list<Stock> li;

  li.push_back(s);   // OK
  li.push_back(ds);  // OUCH!

  return EXIT_SUCCESS;
}
```

STL and Inheritance (2/2)

- Instead, store **smart pointers to heap-allocated objects** in STL containers
  - No slicing! ☺
  - `sort()` does the wrong thing ☹ sorts on addresses by default



Lecture Outline (3/3)

- C++ Inheritance
  - Abstract Classes
  - Static Dispatch
  - Constructors and Destructors
  - Assignment
- **C++ Casts** (Reference: *C++ Primer* §4.11.3, 19.2.1)

Explicit Casting in C

- Simple syntax: `lhs = (new_type) rhs;`
- Used to:
  - Convert between pointers of arbitrary type `(void *) my_ptr`
    - Doesn't change the data, but treats it differently
  - Forcibly convert a primitive type to another `(float) my_int`
    - Actually changes the representation

- You *can* still use C-style casting in C++, but sometimes the intent is not clear
  - You *should not* use C-style casting in C++.

Casting in C++



- C++ provides an alternative casting style that is more informative:
  - `static_cast<to_type>(expression)`
  - `dynamic_cast<to_type>(expression)`
  - `const_cast<to_type>(expression)`
  - `reinterpret_cast<to_type>(expression)`

- Always use these in C++ code
  - Intent is clearer
  - Easier to find in code via searching

Extra Exercise #1

- Design a class hierarchy to represent shapes
  - *e.g.*, Circle, Triangle, Square
- Implement methods that:
  - Construct shapes
  - Move a shape (*i.e.*, add (x,y) to the shape position)
  - Returns the centroid of the shape
  - Returns the area of the shape
  - `Print()`, which prints out the details of a shape

Extra Exercise #2

- Implement a program that uses Extra Exercise #1 (shapes class hierarchy):
  - Constructs a vector of shapes
  - Sorts the vector according to the area of the shape
  - Prints out each member of the vector

- Notes:
  - Avoid slicing!
  - Make sure the sorting works properly!