Poll Everywhere

pollev.com/naomila

Where are you so far on Homework 3?

A. Haven't started yet
B. Working on Part A (Memory-to-File Index Marshaller)
C. Working on Part B (Index Lookup)
D. Working on Part C (File Search Shell)
E. Done!
F. Prefer not to say

CSE333 Systems Programming
C++ Inheritance II

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

- The old exercise swaperoo
  - We take your well-worn Ex8, give you a shiny new Ex9
  - Due next Wednesday (5/20)
  - Building a linked list C++ style, with inheritance!

- Homework 3 is due next Thursday (5/21)
  - Suggestion: Write index files to `/tmp/`, which is a local scratch disk and is very fast, but please clean up when you're done

- Don't try to iron-man next week by doing Ex9 on Tuesday night and Hw3 on Wednesday+Thursday night, okay?
  - Promise  ?

Lecture Outline

- **C++ Inheritance**
  - **Mixed Dispatch**
  - Abstract Classes
  - Constructors and Destructors
  - Assignment

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

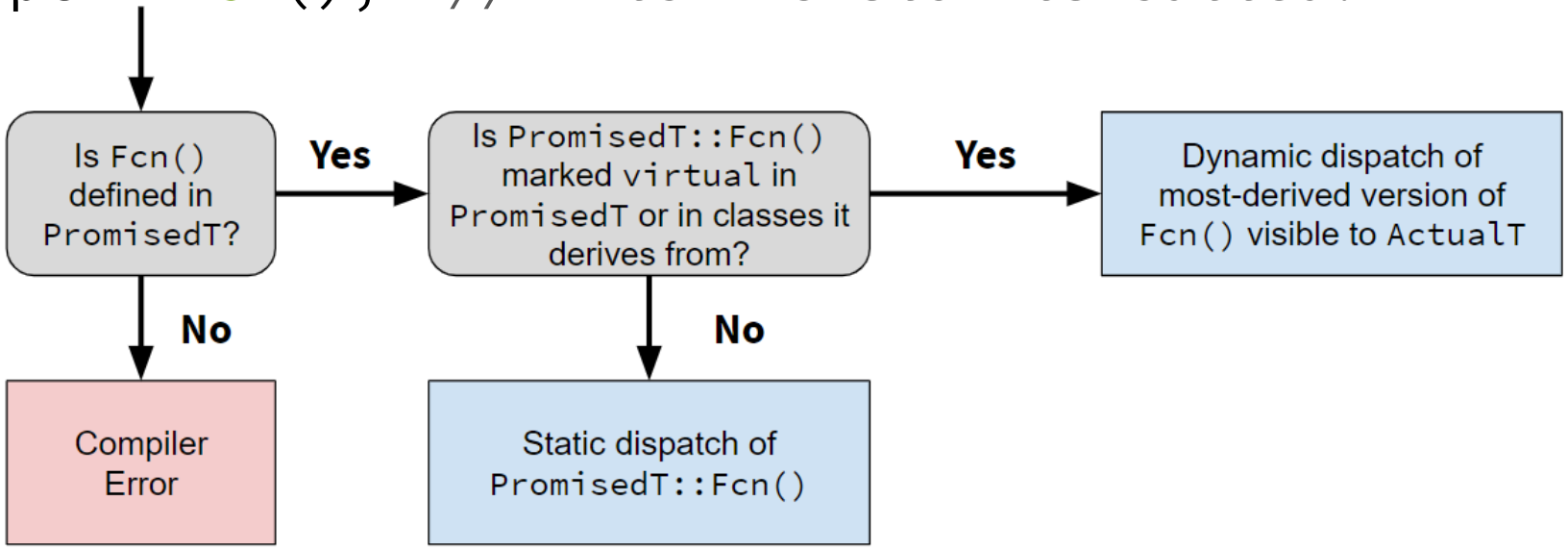
Why Not Always Use `virtual`?

- 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`)

Mixed Dispatch

- Which function is called is a mix of both compile time and runtime decisions as well as *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:

```
PromisedT* ptr = new ActualT;
ptr->Fcn();  // which version is called?
```



Mixed Dispatch Example

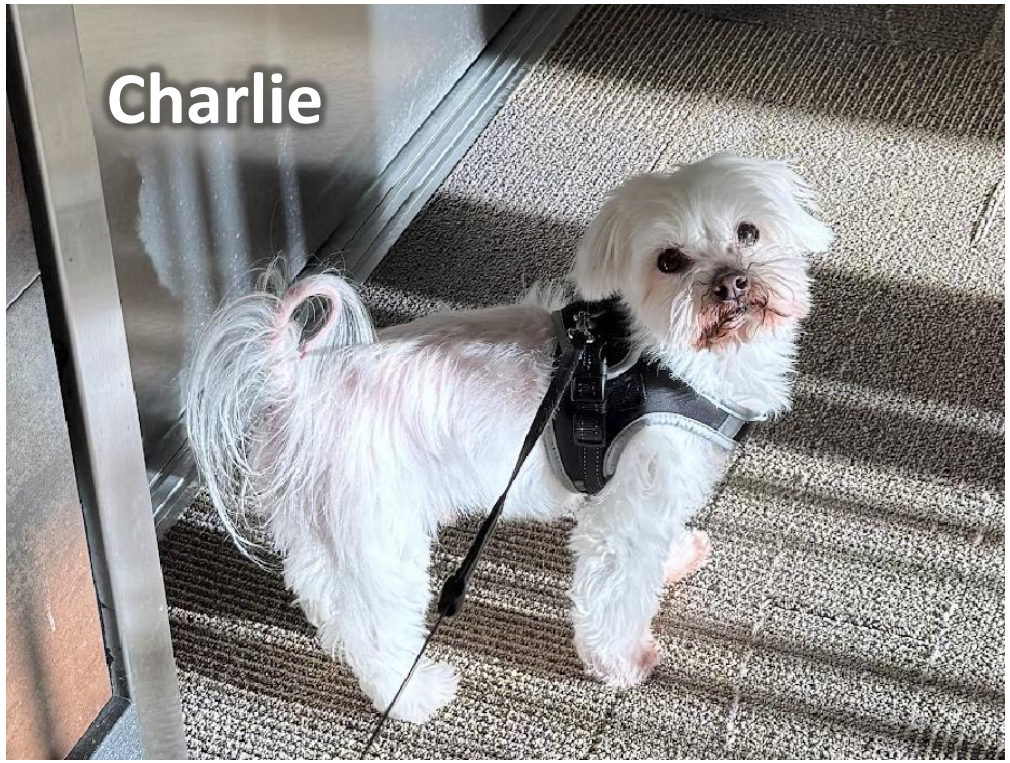
mixed.cc

```
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"; }
};
```

```
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;
  B* b_ptr_b = &b;

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

  a_ptr_b->M1();  //
  a_ptr_b->M2();  //

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

Let's Look at Some Actual Code

- Let's examine the following code using `objdump`
  - `g++ -Wall -g -std=c++17 -o vtable vtable.cc`
  - `objdump -CDS vtable > vtable.d`



vtable.cc

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

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

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

Lecture Outline

- **C++ Inheritance**
  - Mixed Dispatch
  - **Abstract Classes**
  - **Constructors and Destructors**
  - **Assignment**

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

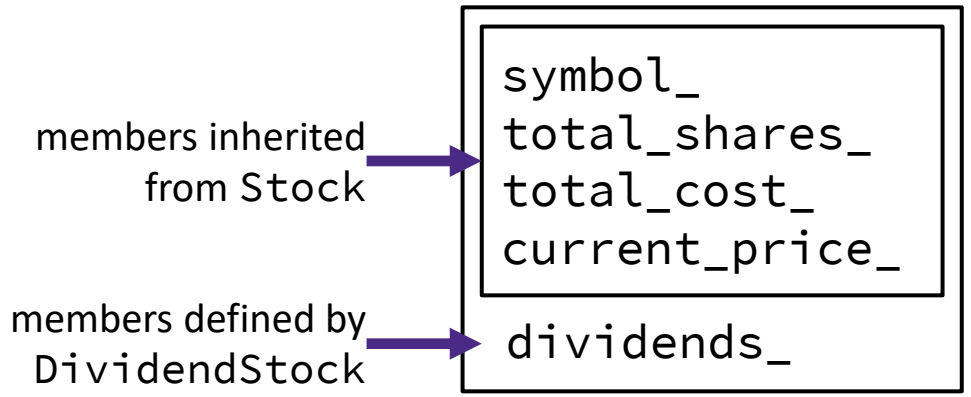
Abstract Classes

- 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

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;
};
```

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

baddtor.cc

```
class Base {
 public:
  Base() { x = new int; }
  ~Base() { delete x; }
  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;  //
  delete b1ptr;  //
}
```

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
  - We need to be careful about assignment operators if we're object slicing
    - See this StackOverflow discussion for details: https://stackoverflow.com/a/14461532

slicing.cc

```
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;  //
  b = d;  //
}
```

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 by default ☹

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!