Poll Everywhere

pollev.com/cse333x

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**

Systems Programming
C++ Inheritance II, Casts

Guest Lecturer:

Naomi Alterman

Teaching Assistants:

Ally Tribble	Blake Diaz	Connor Olson
Grace Zhou	Jackson Kent	Janani Raghavan
Jen Xu	Jessie Sun	Jonathan Nister
Mendel Carroll	Rose Maresh	Violet Monserate

Relevant Course Information

- ❖ Exercise 13 released today, due Monday
    - C++ inheritance: derived and abstract classes

- ❖ Homework 3 is due next Thursday (2/26)
    - Suggestion: Write index files to `/tmp/`, which is a local scratch disk and is very fast, but please clean up when you're done

Lecture Outline (1/4)

- ❖ **C++ Inheritance**
  - ■ **Abstract Classes**
  - ■ **Static Dispatch**
  - ■ Constructors and Destructors
  - ■ Assignment
- ❖ C++ Casting
- ❖ C++ Conversions

- ❖ 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

Reminder: `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

6

What happens if we omit "virtual"?

❖ By default, without `virtual`, methods are dispatched ~~statically~~

- At <u>compile time</u>, 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;
}
```

```
Derived::Foo()
...
```

```
Base::Foo()
...
```

Static Dispatch Example

(handwritten annotations:) impl. in Stock & DivStock
impl in stock inherit by DivStock
calls GetMarketVal

❖ Removed `virtual` on methods:

```cpp
double Stock::GetMarketValue() const;
double Stock::GetProfit() const;
```
*Stock.h*

```cpp
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();
```
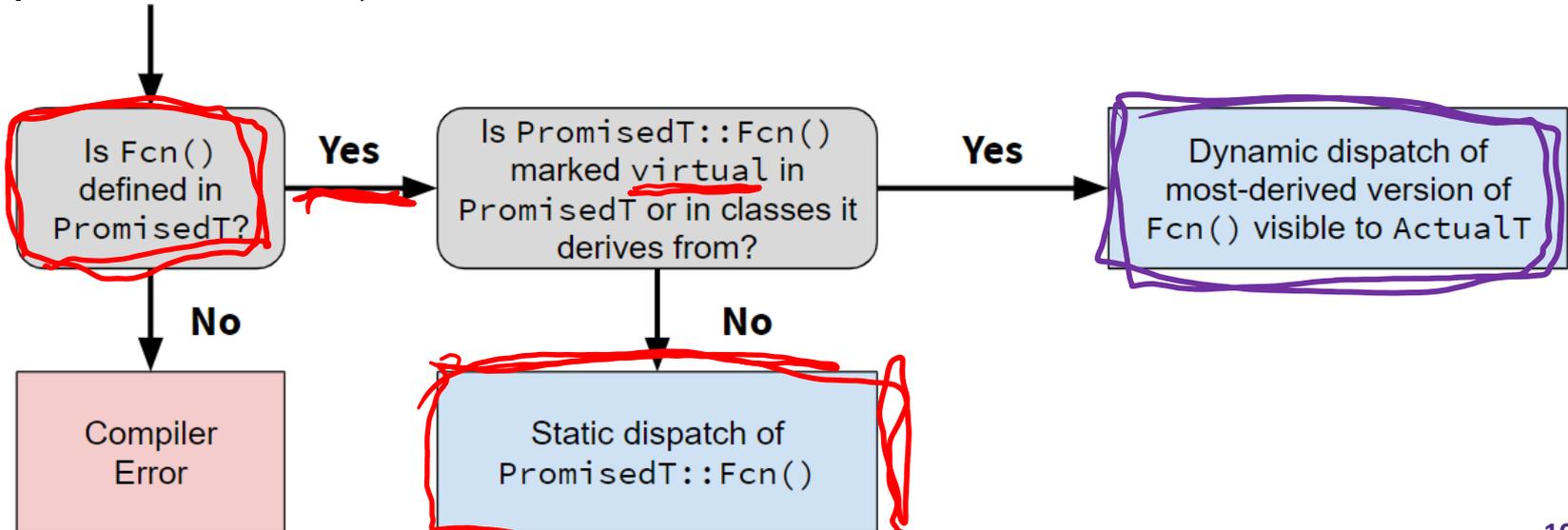
(handwritten diagram: boxes labeled "Div Stock", "ds", "s")

8
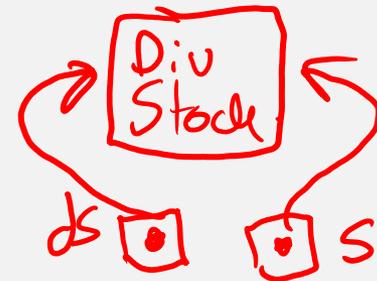
Why Not Always Use `virtual`?

❖ 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?
  ```


Is Fcn() defined in PromisedT?

Yes → Is PromisedT::Fcn() marked virtual in PromisedT or in classes it derives from?

Yes → Dynamic dispatch of most-derived version of Fcn() visible to ActualT

No → Compiler Error

No → Static dispatch of PromisedT::Fcn()

Mixed Dispatch Example

mixed.cc

```cpp
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"; }
};
```

```cpp
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();   // a1
  a_ptr_a->M2();   // a2

  a_ptr_b->M1();   // a1
  a_ptr_b->M2();   // b2

  b_ptr_b->M1();   // b1
  b_ptr_b->M2();   // b2
}
```

Lecture Outline (2/4)

❖ **C++ Inheritance**

- Abstract Classes
- Static Dispatch
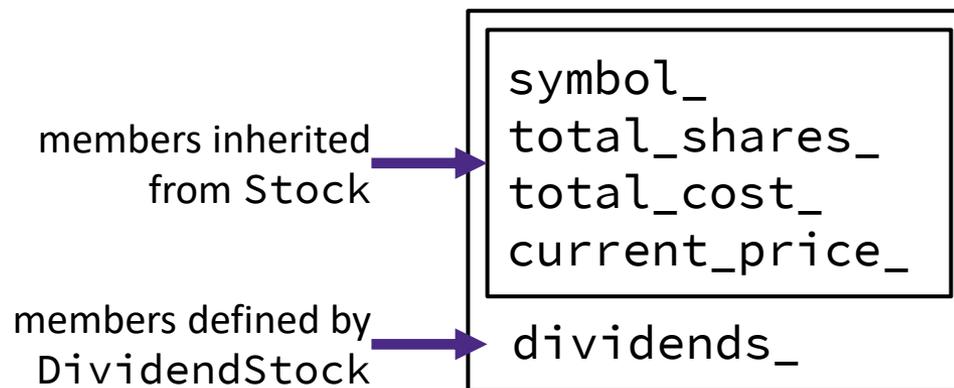- **Constructors and Destructors**
- **Assignment**

❖ C++ Casting

❖ C++ Conversions

❖ Reference: *C++ Primer*, Chapter 15

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:

members inherited
from `Stock` →

```
symbol_
total_shares_
total_cost_
current_price_
```

members defined by
`DividendStock` →

```
dividends_
```

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

goodctor.cc

```cpp
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;
};
```

```cpp
// 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

baddtor.cc

❖ 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

```cpp
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;  //
}
```

LEAK!!!

Assignment and Inheritance

❖ C++ allows you to assign the value of a derived class to an instance of a base class

slicing.cc

```cpp
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;  //
}
```

- 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

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`)
- Ya get sliced ☹ !

```cpp
#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 **pointers to heap-allocated objects** in STL containers

- No slicing! ☺

- **`sort`**`()` does the wrong thing by default ☹

- You have to remember to `delete` your objects before destroying the container ☹

  - Unless you use smart pointers!

Lecture Outline (3/4)

❖ C++ Inheritance
  ▪ Abstract Classes
  ▪ Static Dispatch
  ▪ Constructors and Destructors
  ▪ Assignment

❖ **C++ Casting**

❖ C++ Conversions

❖ 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
  - Doesn't change the data, but treats it differently
- Forcibly convert a primitive type to another
  - 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

staticcast.cc

`static_cast`

```cpp
class A {
 public:
  int x;
};

class B {
 public:
  float x;
};

class C : public B {
 public:
  char x;
};
```

❖ `static_cast` can convert:
- Pointers to classes **of related type**
  - Compiler error if classes are not related
  - Dangerous to cast *down* a class hierarchy
- Casting between `void*` and `T*`
- Non-pointer conversion
  - *e.g.,* `float` to `int`

❖ `static_cast` is checked at <u>compile time</u>

```cpp
void Foo() {
  B b; C c;

  // compiler error
  A* aptr = static_cast<A*>(&b);
  // OK
  B* bptr = static_cast<B*>(&c);
  // compiles, but dangerous
  C* cptr = static_cast<C*>(&b);
}
```

dynamiccast.cc

dynamic_cast

- ❖ dynamic_cast can convert:
  - Pointers to classes **of related type**
  - References to classes **of related type**
- ❖ dynamic_cast is checked at both <u>compile time</u> and <u>run time</u>
  - Casts between unrelated classes fail at compile time
  - Casts from base to derived fail at run time if the pointed-to object is not the derived type

```cpp
class Base {
 public:
  virtual void Foo() { }
  float x;
};

class Der1 : public Base {
 public:
  char x;
};
```

```cpp
void Bar() {
  Base b; Der1 d;

  // OK (run-time check passes)
  Base* bptr = dynamic_cast<Base*>(&d);
  assert(bptr != nullptr);

  // OK (run-time check passes)
  Der1* dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);

  // Run-time check fails, returns nullptr
  bptr = &b;
  dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);
}
```

`const_cast`

❖ `const_cast` adds or strips const-ness
  ▪ Dangerous (**!**)

```cpp
void Foo(int* x) {
  *x++;
}

void Bar(const int* x) {
  Foo(x);                      // compiler error
  Foo(const_cast<int*>(x));  // succeeds
}

int main(int argc, char** argv) {
  int x = 7;
  Bar(&x);
  return EXIT_SUCCESS;
}
```

`reinterpret_cast`

❖ `reinterpret_cast` casts between *incompatible* types
  - Low-level reinterpretation of the bit pattern
  - *e.g.,* storing a pointer in an `int`, or vice-versa
    - Works as long as the integral type is "wide" enough
  - Converting between incompatible pointers
    - Dangerous (**!**)
    - This is used (carefully) in hw3
  - Use any other C++ cast if you can!

Casting Style Considerations

❖ From the "Casting" and "Run-Time Type Information (RTTI)" sections of the Google C++ Style Guide:

- When the logic of a program guarantees that a given instance of a base class is, in fact, an instance of a particular derived class, then a `dynamic_cast` may be used freely on the object.
  - Usually one can use a `static_cast` as an alternative in such situations

- Only use `reinterpret_cast` if you know what you are doing and you understand the aliasing issues
  - For *unsafe conversions* of pointer types to and from integer and other pointer types, including `void*`

Lecture Outline (4/4)

❖ C++ Inheritance
  ▪ Abstract Classes
  ▪ Static Dispatch
  ▪ Constructors and Destructors
  ▪ Assignment
❖ C++ Casting
❖ **C++ Conversions**

❖ Reference: *C++ Primer* §4.11.3, 19.2.1

Implicit Conversion

❖ The compiler tries to infer some kinds of conversions
  ▪ When types are not equal and you don't specify an explicit cast, the compiler looks for an acceptable implicit conversion

```cpp
void Bar(std::string x);

void Foo() {
  int x = 5.7;   // conversion, float -> int
  char c = x;    // conversion, int -> char
  Bar("hi");     // conversion, (const char*) -> string
}
```

Sneaky Implicit Conversions

❖ (`const char*`) to `string` conversion?

- If a class has a constructor with a single parameter, the compiler will exploit it to perform implicit conversions

- At most, one user-defined implicit conversion will happen
  - Can do `int` → `Foo`, but not `int` → `Foo` → `Baz`

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

int Bar(Foo f) {
  return f.x;
}

int main(int argc, char** argv) {
  return Bar(5);  // equivalent to return Bar(Foo(5));
}
```

Avoiding Sneaky Implicits

❖ Declare one-argument constructors as `explicit` if you want to disable them from being used as an implicit conversion path

■ Usually a good idea

```cpp
class Foo {
 public:
  explicit Foo(int xi) : x(xi) { }
  int x;
};

int Bar(Foo f) {
  return f.x;
}

int main(int argc, char** argv) {
  return Bar(5);  // compiler error
}
```

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!