C++ Inheritance II, Casts

CSE 333 Spring 2023

Instructor: Chris Thachuk

Teaching Assistants:

Byron Jin
Deeksha Vatwani
Humza Lala
Noa Ferman
Seulchan (Paul) Han
Tim Mandzyuk
CJ Reith
Edward Zhang
Lahari Nidadavolu
Saket Gollapudi
Timmy Yang
Wui Wu

Relevant Course Information

- Exercise 9 is due next Wednesday (5/17)
- Homework 3 is due next Thursday (5/18)
  - Suggestion: write index files to `/tmp/`, which is a local scratch disk and is very fast, but please clean up when you're done
- Reminder about late days
  - We'll post an updated count of your remaining late days to canvas on Saturday
  - You can find the automatically calculated days used per homework written in a file in Gradescope
  - Can use up to 2 late days per homework (if you have sufficient late days remaining)

Lecture Outline

- **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`

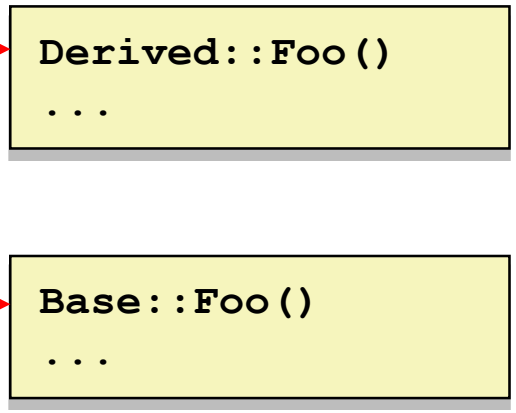
- `F()` will be called using dynamic dispatch even if overridden in a derived class without the `virtual` keyword
  - Good style to help the reader *and avoid bugs* by using `override`
    - Style guide controversy, if you use `override` should you use `virtual` in derived classes? Recent style guides say just use `override`, but you'll sometimes see both, particularly in older code

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:

Stock.h

defined in Stock & DividendStock

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

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();
```

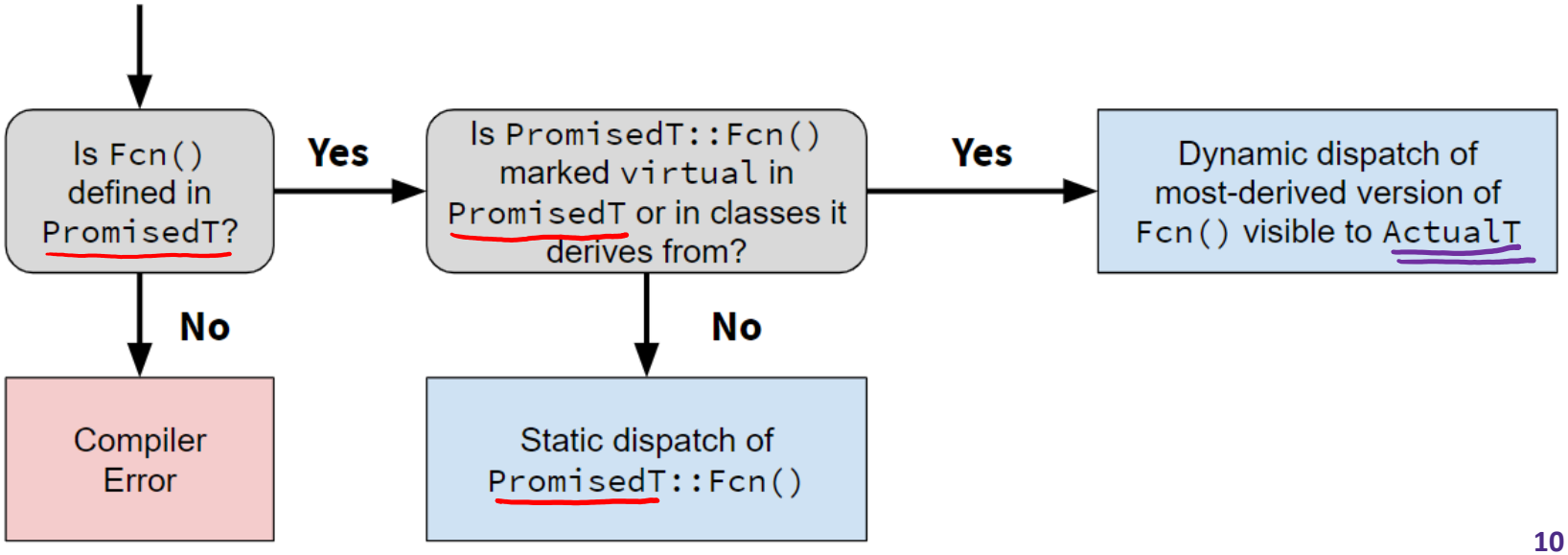
Why Not Always Use `virtual`?

- 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 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"; }
};
```

(virtual)

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

```
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