C++ Inheritance Continued and Casting

CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Audrey Seo

Deeksha Vatwani

Derek de Leuw

Katie Gilchrist

Administrivia

- Congrats on finishing the midterm!

- Exercise 12 was due this morning
- Exercise 13 isn't due until **Monday (August 4th)**
  - Take a break or work on HW3

- HW3 due **next Thursday (August 7th)**

Lecture Outline

- **C++ Inheritance**
  - **Static Dispatch**
  - Abstract Methods and Classes
  - Constructors and Destructors
  - Assignment
- Casting & Conversions
- Introducing: Smart Pointers

Reference: *C++ Primer*, Chapter 15

`virtual` is "sticky"

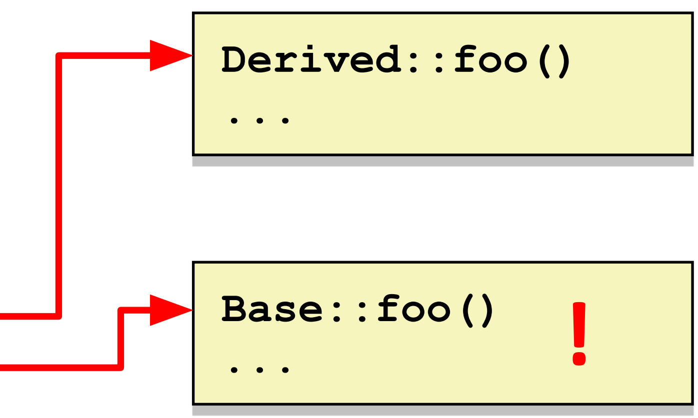
- If `X::f()` is declared virtual, then a vtable will be created for class `X` and for *all* of its subclasses
  - The vtables will include function pointers for (the correct) `f`
- `f()` will be called using dynamic dispatch even if overridden in a derived class without the `virtual` keyword
  - Good 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 <u>compile time</u>, the compiler writes in a `call` to the address of the class' method in the generated code `.text` segment
    - Based on the compile-time visible type of the **pointer**

```
class Base {
            void foo();
};
class Derived : public Base {
  void foo();
};

int main(int argc, char** argv) {
  Derived d;
  Derived* dp = &d;
  Base* bp = &d;
  dp->foo();
  bp->foo();
  return 0;
}
```



Why Not Always Use `virtual`?

- Two (fairly uncommon) reasons:
  - Control:
    - Non-private methods that you want to be **sure** aren't overridden
      - Particularly useful for framework design
  - 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
- In Java, methods are virtual unless specified as `final`
- In C++, methods are static unless specified as `virtual`
  - Omitting virtual can cause hard to understand bugs

Why Not Always Use `virtual`?

- Two (fairly uncommon) reasons:
  - Control:
    - Non-private methods that you want to be **sure** aren't overridden
      - Particularly useful for framework design
  - Eff
    - up)
    - ield

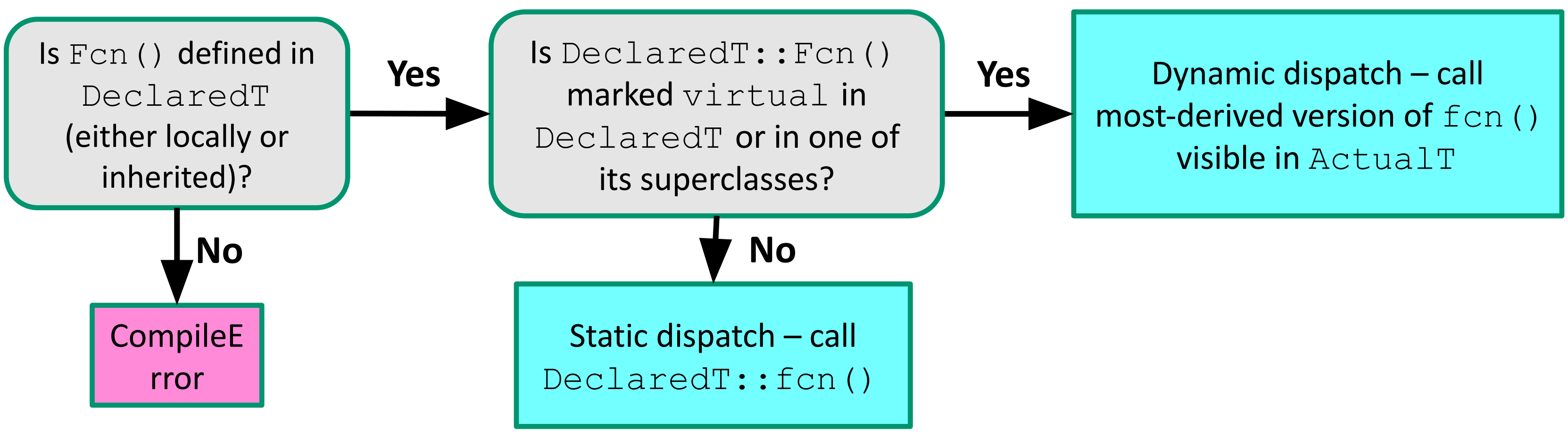
In practice (at least for this class), always use virtual!

- In Java, methods are virtual unless specified as `final`
- In C++, methods are static unless specified as `virtual`
  - Omitting virtual can cause hard to understand bugs

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()`), optimized into a hard-coded function call at compile time (static dispatch)
  - If called via a pointer or reference:

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



Mixed Dispatch Example

mixed.cc

```
class A {
 public:
  void m1() { cout << "a1"; }
  virtual void m2() { cout << "a2"; }
};

class B : public A {
 public:
  void m1() { cout << "b1"; }
  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();   // a1
  a_ptr_a->m2();   // a2

  b_ptr_b->m1();   // b1
  b_ptr_b->m2();   // b2

  a_ptr_b->m1();   // a1
  a_ptr_b->m2();   // b2
}
```

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();  // a1
  a_ptr_a->m2();  // a2

  b_ptr_b->m1();  // b1
  b_ptr_b->m2();  // b2

  a_ptr_b->m1();  // a1
  a_ptr_b->m2();  // b2
}
```

Lecture Outline

- **C++ Inheritance**
  - Static Dispatch
  - **Abstract Methods & Classes**
  - Constructors and Destructors
  - Assignment
- Casting & Conversions
- Introducing: Smart Pointers

Reference: *C++ Primer*, Chapter 15

Abstract Methods

- Sometimes we want to include a method in the interface of a base class but *only* implement it in derived classes
  - In Java, we would use an abstract method
  - In C++, we use a "pure virtual" method
    - Example:

```
virtual string noise() = 0;
```

Abstract Classes

- A class containing *any* pure virtual methods is abstract
  - You can't create instances of an abstract class
  - Derived classes are also abstract unless they override all pure virtual methods
- A class containing *only* pure virtual methods is the same as a Java interface **used to be** (pre-Java 8)
  - Pure type specification without implementations

Lecture Outline

- **C++ Inheritance**
  - Static Dispatch
  - Abstract Methods and Classes
  - **Constructors and Destructors**
  - Assignment
- Casting & Conversions
- Introducing: Smart Pointers

Reference: *C++ Primer*, Chapter 15

Constructors and Inheritance

- A derived class **does not inherit** the base class' constructor
  - The derived class must have its own constructor
  - The base class constructor is automatically invoked *before* the constructor of the derived class

```
class Base {
 public:
  Base() { y = 5; }
  int y;
};

class Der : public Base {
 public:
  Der() { z = y + 3; }
  int z;
};
```

```
int main(void) {
  Der d;
}
```

- First calls Base()
  - Sets y to 5
- Then calls Der()
  - Sets z to 8

Constructors and Inheritance

- If you don't define a any constructors on the derived class, a default constructor will be synthesized (like normal)
- A synthesized default constructor for a derived class:
  - First invokes the default constructor of the base class
  - And then initializes the derived class' member variables

This is okay

```
class Base {
 public:
  Base() : y(5) { }
  int y;
};

class Der : public Base {
 public:
  int z;
};
```

This isn't; compiler error!

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

class Der : public Base {
 public:
  int z;
};
```

Constructors and Inheritance

- If your base class doesn't have a default constructor, you can call a different one using the initialization list
- You can also use this when it **does** have a default constructor, but you want to call a different one.

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

// This works fine
class Der : public Base {
 public:
  Der(int y, int z) : Base(y), z(z) { }
  int z;
};
```

Destructors and Inheritance

- Destructors work similarly
  - Aren't inherited
  - Can be default-synthesized
- But destructors run the base class destructor **after** instead of **before** the derived class destructor

Hint: When in doubt, destructors always run in the reverse order that the constructors ran.

Destructors and Inheritance

- Constructors are always run on a statically-known type
- But destructors can be run on pointer types through `delete`, so dispatch comes into play

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

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

void foo() {
  Base b;
  Der d;
}
```

Destructors and Inheritance

baddtor.cc

- Static dispatch of destructors is almost always a mistake!
  - Good habit to always define a destructor as virtual
  - Here, defining a destructor with an empty body makes sense

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

class Der : public Base {
 public:
  Der() { y = new int; }
            ~Der() { delete y; }
  int* y;
};
void foo() {
  Base* b0ptr = new Base;
  Base* b1ptr = new Der;

  delete b0ptr;  // OK
  delete b1ptr;  // leaks
Der::y
}
```

Lecture Outline

- **C++ Inheritance**
  - Static Dispatch
  - Abstract Methods and Classes
  - Constructors and Destructors
  - **Assignment**
- Casting & Conversions
- Introducing: Smart Pointers

Reference: *C++ Primer*, Chapter 15

Assignment

- In C++, if `A` derives from `B`:
  - We can assign `B*` pointer objects to **`A*`** variables
  - We can assign `B` objects to `A` variables too!

Assignment and Inheritance

- When you assign the value of a derived class to an instance of a base class, it's 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
  - So fields like `y_` get "sliced" off of the object

slicing.cc

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

class Der : public Base {
 public:
  Der(int y) : Base(16), y_(y) { }
  int y_;
};

void foo() {
  Base b(1);
  Der d(2);

  b = d;   // what happens to y_?
  Base b2(d); // same behavior
}
```

Derived-Class Objects

- A derived object contains "subobjects" corresponding to the data members inherited from each base class
  - Fields of the subobject are always next to each other in memory
  - No other 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_`

STL and Inheritance

- 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 0;
}
```

STL and Inheritance

- Instead, store **pointers to heap-allocated objects** in STL containers
  - No slicing! ☺️
  - `sort()` does the wrong thing ☹️
  - You have to remember to `delete` your objects before destroying the container ☹️
    - Smart pointers will help with this!

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Abstract Methods and Classes
  - Constructors and Destructors
  - Assignment
- **Casting & Conversions**
- Introducing: Smart Pointers

- Reference: *C++ Primer*, Chapter 12.1

Explicit Casting in C

- Simple syntax: `lhs = (new_type) rhs;`
- Used in two ways:
  - Convert between pointers of arbitrary types, or between `int`s and pointers
    - Don't change the value, just changes the type
  - Convert one primitive type to another (like rounding `double` to `int`)
    - Actually changes the representation
- You *can* still use C-style casting in C++
  - But it's not as clear what type of casting you're doing

Casting in C++

- C++ provides an alternative casting style that is more informative, with four types:
  - `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

static_cast

staticcast.cc

```
class M {
 public:
  float x;
};

class N : public M {
 public:
  char y;
};

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

- `static_cast` can convert:
  - Pointers or references to classes **of related type**
    - Compiler error if classes are not related
    - Dangerous to cast *down* a class hierarchy
  - Conversion between primitives
    - e.g. `float` to `int`
- `static_cast` is checked at compile time

```
void foo() {
  M m; N n;

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

Use static_cast to cast pointers **up** the class hierarchy, or for numeric casts dynamiccast.cc

dynamic_cast

- dynamic_cast can convert:
  - Pointers or references to classes **of related type**
- dynamic_cast is checked at both compile time and run time
  - Casts between unrelated classes fail at compile time
  - Casts from base to derived return nullptr at run time if the pointed-to object is not the derived type

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

class Der : public Base {
 public:
  ...
```

Use static_cast to cast pointers **down** the class hierarchy, or for casting references

```
void bar() {
  Base b; Der d
  
  // OK (run-time check passes)
  Base* bptr = dynamic_cast<Base*>(&d);
  assert(bptr != nullptr);
  // OK (run-time check passes)
  Der* dptr = dynamic_cast<Der*>(bptr);
  assert(dptr != nullptr);

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

const_cast

- const_cast adds or strips const-ness
  - Dangerous (!)

```
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 0;
}
```

`const_cast`

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

- Can be used (carefully) in certain situations
  - Working with older code that doesn't properly mark read-only functions with `const`
  - Data structures that change internals sometimes without changing the conceptual value (like with caching)

`reinterpret_cast`

- `reinterpret_cast` casts between *incompatible* types
  - Low-level reinterpretation of the bit pattern
  - *e.g.* storing a pointer in an `int64_t`, or vice-versa
    - Works as long as the integral type is "wide" enough
  - Converting between incompatible pointers
    - Dangerous (**!**)
    - This is used (carefully) in hw3

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Abstract Methods and Classes
  - Constructors and Destructors
  - Assignment
- **Casting & Conversions**
  - **Conversions**
- Introducing: Smart Pointers

- Reference: *C++ Primer*, Chapter 12.1

Implicit Conversion

- When expected and actual types are not equal, and you don't specify an explicit cast, the compiler looks for an acceptable implicit conversion

```
void bar(std::string x);

void foo() {
  int x = 5.7;    // conversion, double -> int
  char c = x;     // conversion, int -> char
  bar("hi");      // conversion, (const char*) -> string
}
```

User-defined implicit conversions

- If a class has a constructor with a single parameter, the compiler will use it it to perform implicit conversions
- You can also request it explicitly using `static_cast`
- At most, one user-defined implicit conversion will happen
  - Can do `int` → `Foo`, but not `int` → `Foo` → `Baz`

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

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

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

But `char` → `int` → `Foo` is fine!

Avoiding Accidental Implicit Conversions

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

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

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

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

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Abstract Methods and Classes
  - Constructors and Destructors
  - Assignment
- Casting & Conversions
  - Conversions
- **Introducing: Smart Pointers**

- Reference: *C++ Primer*, Chapter 12.1

Copying in the STL

- Last week we learned about STL, and noticed that STL was doing an enormous amount of copying

- A solution: store pointers in containers instead of objects
  - But this leads to more memory management headaches 💢😨

Manual Memory Management

- In C and C++, we've been manually allocating and deallocating all heap memory

- To do so correctly, we have to think hard about who should free/delete an allocated object
  - **Ownership**: what data structure or code is responsible for freeing data

- This responsibility is mostly *implicit*: it exists in the programmers head
  - Sometimes it will be expressed in comments
  - But not understood by the language or compiler

C++ Smart Pointers

- A smart pointer is an *object* that stores a pointer to heap-allocated data **and** encodes some ideas about ownership
  - A smart pointer looks and behaves like a regular C++ pointer
    - By overloading `*`, `->`, `[]`, etc.
  - The smart pointer will delete the pointed-to object *at the right time* including invoking the object's destructor
    - When that is depends on what kind of smart pointer you use
  - With correct use of smart pointers, you no longer have to remember when to `delete` heap memory!

A Toy Smart Pointer

- We can implement a simple one with:
  - A constructor that accepts a pointer
  - A destructor that frees the pointer
  - Overloaded `*` and `->` operators that access the pointer

ToyPtr Class Template

ToyPtr.h

```
#ifndef TOYPTR_H_
#define TOYPTR_H_

template <typename T> class ToyPtr {
 public:
  explicit ToyPtr(T *ptr) : ptr_(ptr) { }  // constructor
  ~ToyPtr() { delete ptr_; }               // destructor
  T &operator*()  { return *ptr_; }        // * operator
  T *operator->() { return ptr_;  }        // -> operator

 private:
  T *ptr_;                                 // the pointer
};

#endif  // TOYPTR_H_
```

This is weird! The overload for the -> operator behaves differently than others

ToyPtr Example

usetoy.cc

```
#include <iostream>
#include "ToyPtr.h"

// simply struct to illustrate the "->" operator
struct Point { int x = 1; int y = 2; };
std::ostream &operator<<(std::ostream &out, const Point &rhs) {
  return out << "(" << rhs.x << "," << rhs.y << ")";
}

int main(int argc, char **argv) {
  // Create a dumb pointer
  Point *leak = new Point;

  // Create a "smart" pointer
  ToyPtr<Point> notleak(new Point);

  std::cout << "     *leak: " << *leak << std::endl;
  std::cout << "   leak->x: " << leak->x << std::endl;
  std::cout << "  *notleak: " << *notleak << std::endl;
  std::cout << "notleak->x: " << notleak->x << std::endl;

  return 0;
}
```

ToyPtr Example

usetoy.cc

```
#include <iostream>
#include "ToyPtr.h"
```

```
==2554== Memcheck, a memory error detector
==2554== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==2554== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==2554== Command: ./usetoy
==2554==
      *leak: (1,2)
   leak->x: 1
  *notleak: (1,2)
notleak->x: 1
==2554==
==2554== HEAP SUMMARY:
==2554==     in use at exit: 8 bytes in 1 blocks
==2554==   total heap usage: 4 allocs, 3 frees, 74,768 bytes allocated
```

```
  std::cout << "      *leak: " << *leak << std::endl;
  std::cout << "    leak->x: " << leak->x << std::endl;
  std::cout << "   *notleak: " << *notleak << std::endl;
  std::cout << "notleak->x: " << notleak->x << std::endl;

  return 0;
}
```

What Makes This a Toy?

- Can’t handle:
  - Arrays
  - Copying
  - Reassignment
  - Comparison
  - … plus many other subtleties…

- Luckily, others have built non-toy smart pointers for us!

Administrivia

- Check your HW1 grades
  - If you got a zero and you turned it in, it's likely a tagging issue. File a regrade request!
- Exercise 13 isn't due until **Monday (August 4th)**
  - Take a break or work on HW3
- HW3 due **next Thursday (August 7th)**

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!