Inheritance & Casting

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat	Jialin Cai
Mati Davis	Donglun He
Chandravaran Kunjeti	Heyi Liu
Shufan Liu	Eddy Yang

Poll Everywhere

pollev.com/tqm

- Any questions from previous lectures?

Logistics

- HW4 Posted Due Thursday 4/20 @ 11:59

- Project Released! Due Wednesday 4/26 @ 11:59

- Travis has extra Office Hours from 10:15 am to 12:15 pm this Thursday 4/13

Logistics

- Final Exam Scheduling:
  - 96 hours (4 days)
  - Opens Tuesday May 2nd @ Noon
  - Closes Saturday May 6th @ noon

Lecture Outline

- **C++ Inheritance**
  - **Static Dispatch**
  - Constructors and Destructors
  - Assignment
- C++ Casting

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

Dynamic Dispatch (like Java)

- Usually, when a derived function is available for an object, we want the derived function to be invoked
  - This requires a *run time* decision of what code to invoke
- A member function invoked on an object should be the *most-derived function* accessible to the object's visible type
  - Can determine what to invoke from the *object* itself
- Example:
  - `void PrintStock(Stock* s) { s->Print(); }`
  - Calls the appropriate `Print()` without knowing the actual type of `*s`, other than it is some sort of `Stock`

Is this a Stock or a DividendStock ?

Requesting Dynamic Dispatch (C++)

- Prefix the member function declaration with the `virtual` keyword
  - Derived/child functions don't need to repeat `virtual`, but was traditionally good style to do so
  - This is how method calls work in Java (no virtual keyword needed)
  - You almost always want functions to be virtual

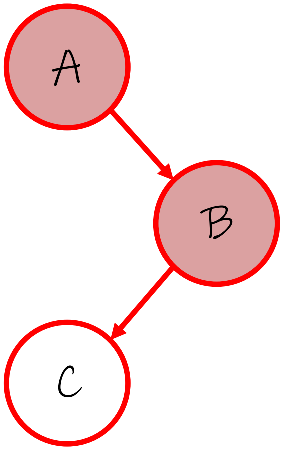
Reminder: `virtual` is "sticky"

- 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

Most-Derived

```
class A {
 public:
  // Foo will use dynamic dispatch
  virtual void Foo();
};

class B : public A {
 public:
  // B::Foo overrides A::Foo
  void Foo();
};

class C : public B {
  // C inherits B::Foo()
};
```

```
void Bar() {
  A* a_ptr;
  C c;

  a_ptr = &c;

  // Whose Foo() is called?
  a_ptr->Foo(); // B::Foo
}
```



Has Foo definition

Poll Everywhere

pollev.com/tqm

- Whose `Foo()` is called?

```
void Bar() {
  A* a_ptr;
  A a;
  D d;

  // Q1:
  a_ptr = &a;
  a_ptr->Foo();

  // Q2:
  a_ptr = &d;
  a_ptr->Foo();
}
```

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

class B : public A {
 public:
  void Foo();
};

class C : public B {
};

class D : public C {
 public:
  void Foo();
};

class E : public C {
};
```

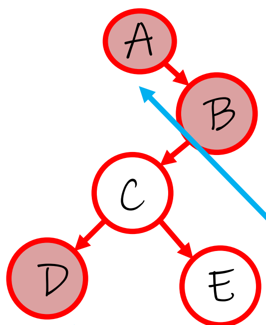
	Q1	Q2
A.	A	B
B.	A	D
C.	B	B
D.	B	D

E. We're lost…

Poll Everywhere

pollev.com/tqm

- Whose `Foo()` is called?



	Q1	Q2
A.	A	B
B.	A	D
C.	B	B
D.	B	D
E.	We're lost...	

```
void Bar() {
  A* a_ptr;
  A a;
  D d;

  // Q1:
  a_ptr = &a;
  a_ptr->Foo();
        A::Foo()
  // Q2:
  a_ptr = &d;
  a_ptr->Foo();
        D::Foo()
}
```

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

class B : public A {
 public:
  void Foo();
};

class C : public B {
};

class D : public C {
 public:
  void Foo();
};

class E : public C {
};
```

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;
}
```

`dp->foo();` →

```
Derived::foo()
...
```

`bp->foo();` →

```
Base::foo()
...
```

Poll Everywhere

pollev.com/tqm

- Whose `Foo()` is called?

	Q1	Q2
A.	A	B
B.	A	D
C.	D	B
D.	D	D
E.	We're lost...	

```
void Bar() {
  D d;

  A* a_ptr = &d;
  C* c_ptr = &d;

  // Q1:
  a_ptr->Foo();

  // Q2:
  c_ptr->Foo();
}
```

test.cc

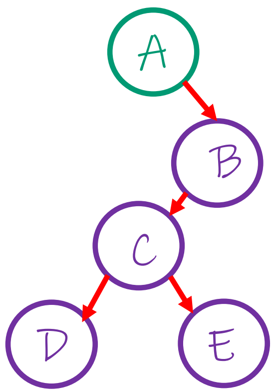
```
class A {
 public:
   void Foo();
};

class B : public A {
 public:
  virtual void Foo();
};

class C : public B {
};

class D : public C {
 public:
  void Foo();
};

class E : public C {
};
```

Poll Everywhere

pollev.com/tqm

- Whose **Foo**() is called?

test.cc



Key:
Static dispatch
Dynamic dispatch

	Q1	Q2
A.	A	B
B.	A	D
C.	D	B
D.	D	D
E.	We're lost...	

```
void Bar() {
  D d;

  A* a_ptr = &d;
  C* c_ptr = &d;

  // Q1: A::foo
  a_ptr->Foo();

  // Q2: D::foo
  c_ptr->Foo();
}
```

```
class A {
 public:
   void Foo();
};

class B : public A {
 public:
  virtual void Foo();
};

class C : public B {
};

class D : public C {
 public:
  void Foo();
};

class E : public C {
};
```

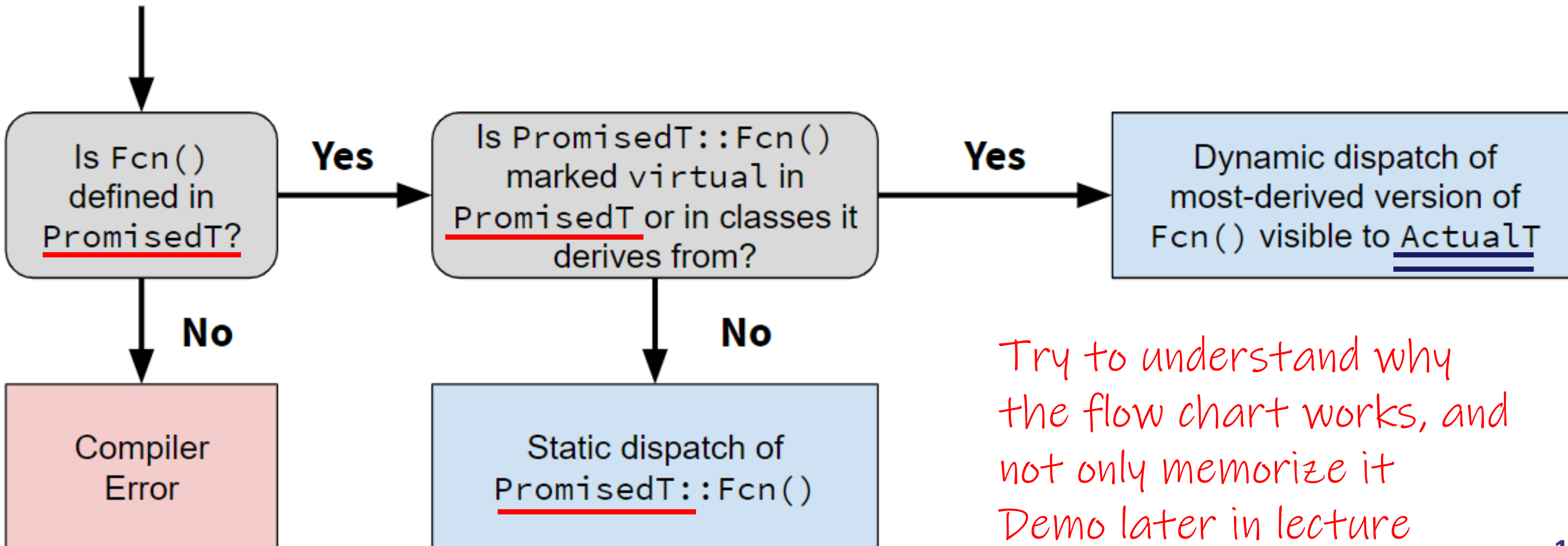
Why Not Always Use `virtual`?

- 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`)

Dispatch Decision Tree

- 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?
```



Try to understand why the flow chart works, and not only memorize it
Demo later in lecture

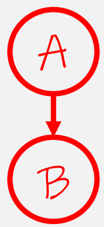
Mixed Dispatch Example

Key:
Static dispatch
Dynamic dispatch

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
(remember, virtual is "sticky")




```
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::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
}
```

promisedType
actualType
Compiler error (B* b_ptr_a = &a; struck out)

Poll Everywhere

pollev.com/tqm

- Apply what you've learned to a more complex example!
- What is printed?

A. HI
B. HA
C. Compiler Error
D. Segmentation fault
E. We're lost…

```
int main() {
  B b;
  B* b_ptr = &b;

  // Q:
  b_ptr->Foo();
}
```

poll.cc

```
class A {
 public:
  virtual void Foo() {
    cout << "H";
    this->Bar();
  }

  void Bar() {
    cout << "A";
  }
};

class B : public A {
 public:
  virtual void Bar() {
    cout << "I";
  }
};
```

Poll Everywhere

pollev.com/tqm

- Apply what you've learned to a more complex example!
- What is printed?

"this"
is of type A*
in this context
So, static dispatch

A. HI

B. HA

C. Compiler Error

D. Segmentation fault

E. We're lost...

If we removed "this->"
we would get same behaviour

poll.cc

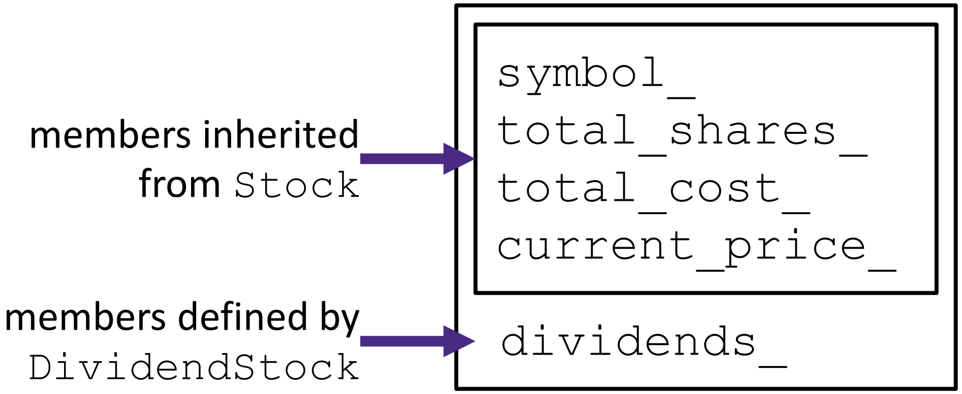
```
class A {
 public:
  virtual void Foo() {
    cout << "H";
    this->Bar();
  }

  void Bar() {
    cout << "A";
  }
};

class B : public A {
 public:
  virtual void Bar() {
    cout << "I";
  }
};
```

```
int main() {
  B b;
  B* b_ptr = &b;

  // Q:
  b_ptr->Foo();
}
```

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 sub-objects)
  - Base sub-object ***usually*** first in memory on Linux

- Conceptual structure of `DividendStock` object:



Demo: From structs to objects

- See `static_dispatch/`
  - How do you properly handle memory management?
    - Ignores dynamic dispatch for now

- See `dynamic_dispatch/`
  - Rewriting static_dispatch code to hold vtables and dynamic dispatch?

Lecture Outline

- **C++ Inheritance**
  - Static Dispatch
  - **Constructors and Destructors**
  - **Assignment**
- C++ Casting

- 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
  - A synthesized default constructor for the derived class first invokes the default constructor of the base class and then initialize 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;
};
```

Compiler error ☹ No default ctor

Invokes a specific ctor

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;
};
```

Because base has default ctor

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's x
  delete b1ptr;   // delete's x, but not y
}
```

Not virtual, Static dispatch

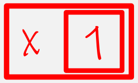
Both invoke Base dtor!!!!

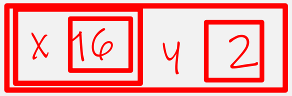
Assignment and Inheritance

- 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`

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;   // Compiler error – not enough info
  b = d;   // ok, What happens to y?
               Y is not copied over.
}
```





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 EXIT_SUCCESS;
}
```

STL and Inheritance

- Instead, store **pointers to heap-allocated objects** in STL containers
  - No slicing! ☺ Vector<Stock*>
  - `sort()` does the wrong thing ☹ Sorts by address value on default
  - You have to remember to `delete` your objects before destroying the container ☹
    - Unless you use Smart pointers!

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Constructors and Destructors
  - Assignment
- **C++ Casting**

- 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 — (double) my_int
    - Actually changes the representation
- You *can* still use C-style casting in C++, but sometimes the intent is not clear

Casting in C++

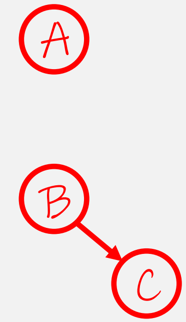
- C++ provides an alternative casting style that is more informative:
  - static_cast<to_type>(expression)
  - dynamic_cast<to_type>(expression)
  - const_cast<to_type>(expression)
  - reinterpret_cast<to_type>(expression)

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

static_cast

staticcast.cc

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

class B {
 public:
  float y;
};

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



Any well-defined conversion

- `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 `void*` to `T*`
  - Non-pointer conversion
    - *e.g.* `float` to `int`
- `static_cast` is checked at <u>compile time</u>

```
void foo() {
  B b; C c;

  // compiler error   Unrelated types
  A* aptr = static_cast<A*>(&b);
  // OK   Would have worked without cast
  B* bptr = static_cast<B*>(&c);
  // compiles, but dangerous
  C* cptr = static_cast<C*>(&b);
}    What happens when you do cptr->z?
```

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 compile time and run time
  - 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
- Can be used like `instanceof` from java

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

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

```
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 (!)

```
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 (**!**)
  - Use any other C++ cast if you can.