C++ Inheritance II, Casting
CSE 333 Spring 2018

Instructor: Justin Hsia

Teaching Assistants:
Danny Allen        Dennis Shao        Eddie Huang
Kevin Bi           Jack Xu            Matthew Neldam
Michael Poulain    Renshu Gu         Robby Marver
Waylon Huang        Wei Lin
Administrivia

- Exercise 14 released today, due Friday
  - C++ inheritance with abstract class

- hw3 is due next Thursday (5/17)
  - Section tomorrow will also help you get started

- Midterm grading
  - Submit regrade requests via Gradescope for each subquestion
    - These go to different graders
  - Regrade requests open until end of tomorrow (5/10)
  - Exam will be curved up (free points for everyone!)
Lecture Outline

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

- **C++ Casting**

- 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 by using **override** and **virtual** in derived classes
Static (Non-Virtual) Dispatch

- By default, methods are dispatched \textit{statically}
  - At \textit{compile time}, the compiler writes in a \texttt{call} to the address of the class’ method in the \texttt{.text} segment
    - Based on the compile-time visible type of the callee
  - This is \textit{different} than Java

```cpp
class Derived : public Base { ...
};

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

```c
Derived::\texttt{foo}()  
add $0x1d, %eax ...

Base::\texttt{foo}()  
add $0x1b, %eax ...
```
Static Dispatch Example

- **Removed** `virtual` on methods:

```cpp
Stock.h

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

```cpp
DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;

// Invokes DividendStock::GetMarketValue()
ds->GetMarketValue();

// Invokes Stock::GetMarketValue()
s->GetMarketValue();

// invokes Stock::GetProfit(), since that method is inherited.
// Stock::GetProfit() invokes Stock::GetMarketValue().
ds->GetProfit();

// invokes Stock::GetProfit().
// Stock::GetProfit() invokes Stock::GetMarketValue().
s->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
Mixed Dispatch Example

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;  // compiler error
    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
}
Peer Instruction Question

- Whose `Foo()` is called?

Q1  Q2

A. A   A
B. A   B
C. D   A
D. D   B
E. We’re lost…

```cpp
class A {
    public:
    void Foo();   //static
};
class B : public A {
    public:
    virtual void Foo();
};   // dynamic

class C : public B {
};
class D : public C {
    public:
    (virtual)void Foo();   //dynamic
};
class E : public C {
};

void Bar() {
    D d;
    E e;
    A* a_ptr = &d;
    C* c_ptr = &e;

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

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

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

- C++ Casting

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:
Constructors and Inheritance

- A derived class **does not inherit** the base class’ constructor
  - The derived class must have its own constructor
  - The synthesized default constructor will initialize the derived class’ non-“plain ‘ol data” member variables to zero-equivalents and invokes the default constructor of the base class
    - 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

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

goodctor.cc

```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 z) : z(z) { }
    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

```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; // OK
    delete b1ptr; // leaks Der1::y
}
```
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

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

class Der1 : public Base {
    public:
        Der1(int y) : Base(16), y(y) { }
        int y;
};

void foo() {
    Base b(1);
    Der1 d(2);

    d = b; // compiler error - not enough info
    b = d; // OK, but what happens to y?
}
```
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 😞

```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 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!
Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Abstract Classes
  - Constructors and Destructors
  - Assignment

- C++ Casting

Reference:  *C++ Primer* §4.11.3, 19.2.1
Explicit Casting in C

- Simple syntax: \( \text{lhs} = (\text{new\_type}) \text{rhs}; \)

- Used to:
  - Convert between pointers of arbitrary type
    - Don't change the data, but treat 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
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**

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

- **static_cast** is checked at compile time

```cpp
class A {
    public:
    int x;
};
class B {
    public:
    float x;
};
class C : public B {
    public:
    char x;
};

void foo() {
    B b; C c;

    // compiler error (unrelated)
    A* aPtr = static_cast<A*>(&b);

    // OK (would have been done implicitly)
    B* bPtr = static_cast<B*>(&c);

    // compiles, but dangerous
    C* cPtr = static_cast<C*>(&b);
}
```
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 a full derived object

---

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 (!)

```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 0;
}
```
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
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
    bar("hi");   // conversion, (const char*) -> string
    char c = x;   // conversion, int -> char
}
```
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 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));
}
```
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 x) : x(x) {}
        int x;
    };

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

int main(int argc, char** argv) {
    return Bar(5);  // compiler error — no longer allowed
}
```
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
  - \texttt{Print()} \textcolor{red}{\textit{()}, 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!