CSE 374
Programming Concepts & Tools

Brandon Myers

Winter 2015

Lecture 24 – C++ Subclasses and Inheritance

(Thanks to Hal Perkins)

Subclassing

- In many ways, Object-oriented programming (OOP) is “all about” subclasses overriding methods
  - Often not what you want, but what makes OOP fundamentally different from, say, functional programming (Scheme, ML, Haskell, etc., cf. CSE413)
- C++ gives you lots more options than Java with different defaults, so it’s easy to scream “compiler bug” when you mean “I’m using the wrong feature”…

Subclassing in C++

- Basic subclassing:

```
class D : public C { ... }
```

- This is *public inheritance*; C++ has other kinds too (won't cover)
  - Differences affect visibility and issues when you have multiple superclasses (won't cover)
  - So **do not forget** the `public` keyword

More on subclassing

- Not all classes have superclasses (unlike Java with the **Object** class)
  - (and classes can have multiple superclasses — more general and complexity-prone than Java)
- Terminology
  - Java (and others): “superclass” and “subclass”
  - C++ (and others): “base class” and “derived class”
- Our example code: **`House`** derives from **`Land`** which derives from **`Property`** (read the code, no time for detailed presentation)
- As in Java, can add fields/methods/constructors, and override methods

Constructor and destructors

- Constructor of base class gets called before constructor of derived class
  - Default (zero-argument) constructor unless you specify a different one after the `:` in the constructor
  - Initializer syntax:
    `Foo::Foo(…): Bar(args); it(x) { … }`
    - Needed to execute superclass constructor with arguments; also works on instance variables and is preferred in production code (slogan: “initialization preferred over assignment”)
- Destructor of base class gets called after destructor of derived class
- So constructors/destructors really extend rather than override, since that is typically what you want
  - Java is the same

Method overriding, part 1

- If a derived class defines a method with the same method name and argument types as one defined in the base class (perhaps because of an ancestor), it *overrides* (i.e., replaces) rather than *extends*
- If you want to use the base-class code, you specify the base class when making a method call (**`class::method(…)`**)
  - Like **`super`** in Java (no such keyword in C++ since there may be multiple inheritance)
- NOTE: the title of this slide is *part 1*
  - (more later)

Casting and subtyping

- An object of a derived class *cannot* be cast to an object of a base class.
  - For the same reason a `struct T1 {int x,y,z;}` cannot be cast to type `struct T2 {int x, y;}` (different size)
- A pointer to an object of a derived class *can* be cast to a pointer to an object of a base class.
  - For the same reason a `struct T1*` can be cast to type `struct T2*` (pointers to a location in memory)
  - (Story not so simple with multiple inheritance)
- After such an *upcast*, field-access works fine (prefix), but what do method calls mean in the presence of overriding?

An important example

```
class A {
  public:
    void m1() { cout << "a1" << endl; }
    virtual void m2() { cout << "a2" << endl; }
};
class B : public A {
  public:
    void m1() { cout << "b1" << endl; }
    void m2() { cout << "b2" << endl; }
};

int main() {
  B* b = new B();
  A* a = b;
  a->m1();
  a->m2();
  b->m1();
  b->m2();
}
```

In words…

- A non-virtual method-call is *resolved* using the (compile-time) type of the *receiver* expression
- A virtual method-call is *resolved* using the (run-time) class of the *receiver* object (what the expression evaluates to)
  - Like in Java
  - Called "dynamic dispatch"
- A method-call is virtual if the method called is marked **`virtual`** or overrides a virtual method
  - So "one virtual" somewhere up the base-class chain is enough, but it's probably better style to repeat it