CSE 303:
Concepts and Tools for Software Development

Dan Grossman

Spring 2007

Lecture 24— C++ continued

In the middle of C++

Doing a tiny fraction of an enormous language:

- Many small conveniences over C
- OOP without everything being a pointer to a heap object
- OOP with manual memory management and lots of HYCSBWK things
- OOP with different kinds of inheritance and overriding

Back to our first class-definition in `Property.h`, `Property.cc`...

OOP in C++, part 1

Like Java:

- Fields vs. methods, static vs. instance, constructors
- Method overloading (functions, operators, and constructors too)

Not quite like Java:

- access-modifiers (e.g., `private`) syntax and default
- declaration separate from implementation (like C)
- funny constructor syntax, default parameters (e.g., `... = 0`)

Nothing like Java:

- Objects vs. pointers to objects
- Destructors and copy-constructors
- virtual vs. non-virtual (to be discussed)

Stack vs. heap

Java: cannot stack-allocate an object (only a pointer to one).

C: can stack-allocate a struct, then initialize it.

C++: stack-allocate and call a constructor (where `this` is the object's address, as always)

- `Property p1(10000);`

Java: `new Property(...)` calls constructor, returns heap-allocated pointer.

C: Use `malloc` and then initialized, must free exactly once later.

C++: Like Java, but can also do `new int(42)`. Like C must deallocate, but must use `delete` instead of `free`.

Destructors

An object's *destructor* is called just before the space for it is reclaimed.

A common use: Reclaim space for heap-allocated things pointed to (first calling their destructors).

- But not if there are other pointers to it (*aliases*)?!

Meaning of `delete x`: call the destructor of pointed-to heap object, then reclaim space.

Destructors also get called for stack-objects (when they leave scope).

Advice: Always make destructors `virtual` (learn why soon)

Arrays

Create a heap-allocated array of objects: `new A[10];`

- Calls *default* (zero-argument) constructor for each element.
- Convenient if there's a good default initialization.

Create a heap-allocated array of pointers to objects: `new A*[10]`

- More like Java (but not initialized?)
- As in C, `new A()` and `new A[10]` have type `A*`.
- `new A*` and `new A*[10]` both have type `A**`.
- Unlike C, to delete a non-array, you must write `delete e`
- Unlike C, to delete an array, you must write `delete [] e`

Else HYCSBWK – the deleter must know somehow what is an array.

Digression: Call-by-reference

In C, we know function arguments are *copies*

- But copying a pointer means you still point to the same (uncopied) thing

Same in C++, but a “reference parameter” (the & character after it) is different.

Callee writes: `void f(int& x) { x = x + 1; }`

Caller writes: `f(y)`

But it’s *as though* the caller wrote `f(&y)` and everywhere the callee said `x` they really said `*x`.

So that little & has a big meaning.

Copy Constructors

In C, we know `x=y` or `f(y)` copies `y` (if a struct, then member-wise copy).

Same in C++, unless a *copy-constructor* is defined, then do *whatever it says*.

A copy-constructor by definition takes a reference parameter (else we'd need to copy, but that's what we're defining) of the same type.

Let's not talk about the `const`.

Our example use is strange (why increment a counter), but useful for understanding what happens.

Now more OOP: Subclassing

To me, OOP is "all about" subclasses overriding methods.

- Often not what you want, but what makes OOP fundamentally different from, say, functional programming (CSE341)

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"...

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 `Object`)
- I prefer terms "superclass" and "subclass" but C++ programmers tend to use "base class" and "derived class"
  - Just a terminology thing
- Our example code: `House` derives from `Land` which derives from `Property`
- As in Java, can add fields/methods/constructors, and override methods.

Construction and destruction

- Constructor of base class gets called *before* constructor of derived class
  - Default (zero-arg) constructor unless you specify a different one after the : in the constructor.
- 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.

Method overriding, part 1

If a derived class defines a method with the same 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.

- Like `super` in Java (no such keyword in C++ since there may be multiple inheritance)

Warning: the title of this slide is *part 1*.