CSE 303:
Concepts and Tools for Software Development

Hal Perkins
Winter 2009
Lecture 24—Introduction to C++
C++ is an enormous language:

- All of C
- Classes and objects (kind of like Java, some crucial differences)
- Many more little conveniences (I/O, new/delete, function overloading, pass-by-reference, bigger standard library)
- Namespaces (kind of like Java packages)
- Stuff we won’t do: const, different kinds of casts, exceptions, templates, multiple inheritance, . . .

We will focus on a couple themes rather than just a “big bag of new features to memorize” . . .
Our focus

OOP in a C-like language may help you understand C and Java better?

- We can put objects on the stack or the heap; an object is not a pointer to an object
- Still have to manage memory manually
- Still lots of ways to HCBWKMSCOD
- Still distinguish header files from implementation files
- Allocation and initialization still separate concepts, but easier to “construct” and “destruct”
- Programmer has more control on how method-calls work (different defaults from Java)

\[a\text{(hopefully crash, but who knows – might silently corrupt other data)}\]
Hello World

#include <iostream>
int main() {
    // Use standard output stream cout
    // and operator << to send "Hello World"
    // and an end line to stdout
    std::cout << "Hello World" << std::endl;
    return 0;
}

Differences from C: “new-style” headers (no .h), namespace access (::), I/O via stream operators, ...

Differences from Java: not everything is in a class, any code can go in any file, ...
Compiling

Need a different compiler than for C; use g++ on attu. Example:

```bash
$ g++ -Wall -o hello hello.cc
```

The `.cc` extension is a convention (just like `.c` for C), but less universal (also see `.cpp`, `.cxx`, `.C`).

Uses the C preprocessor (no change there).

Now: A few “niceties” before our real focus (classes and objects).
I/O

Operator \texttt{\textless\textless} takes a “ostream” and (various things) and outputs it; returns the stream, which is why this works:
\begin{verbatim}
std::cout \texttt{\textless\textless} 3 \texttt{\textless\textless} "hi" \texttt{\textless\textless} f(x) \texttt{\textless\textless} \'\n';
\end{verbatim}

- Easier and safer than printf

Operator \texttt{\textgreater\textgreater} takes “istream” and (various things) and inputs into it.

- Easier and safer than scanf. Do \textit{not} use pointers; e.g.,
\begin{verbatim}
int x; std::cin \texttt{\textgreater\textgreater} x;
\end{verbatim}

Can “think of” \texttt{\textgreater\textgreater} and \texttt{\textless\textless} as keywords, but they are not:

- \textit{Operator overloading} redefines them for different pairs of types.
  - In C they mean “left-shift” and “right-shift” (of bits);
    undefined for non-numeric types.

- Lack of address-of for input done with \textit{call-by-reference} (later).
Namespaces

In C, all non-static functions in the program need different names

• Even operating systems with tens of millions of lines.

Namespaces (cf. Java packages) let you group top-level names:

• namespace myspace { ... definitions ... }

• Of course, then different namespaces can have the same function names and they are totally different functions.

• Can nest them

• Can reuse the same namespace in multiple places
  – Particularly common: in the .h and the .cc

For example, the whole C++ standard library is in namespace std.

To use a function/variable/etc. in another namespace, do thespace::someFun() (not . like in Java)
Using

To avoid having to write namespaces and :: constantly, use a *using declaration*

Example:

```cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World" << endl;
    return 0;
}
```
Onto OOP

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?)
- `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.
Now more OOP: Subclassing

In many ways, 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)
  – (and classes can have multiple superclasses — more general and complexity-prone than Java, where a class has one superclass and can also implement interfaces)

• 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

• 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.

- Java is the same