CSE 374
Programming Concepts & Tools

Hal Perkins
Winter 2013
Lecture 19 – 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

Object-oriented programming 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)

*hopefully crash, but who knows – might silently corrupt other data
References

• Lectures and sample code will have enough to get by for cse374

• Beyond that, best place to start: C++ Primer, Lippman, Lajoie, Moo, 5th ed., Addison-Wesley, 2013

• Every serious C++ programmer should also read Effective C++, Meyers, 3rd ed., Addison-Wesley, 2005

• Good online source: cplusplus.com
Hello World

```cpp
#include <iostream>
int main() {
    // Use standard output stream cout
    // and operator << to send "Hello World"
    // and a newline (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, ...
  - Can write procedural programs if that’s what you want
Compiling

• Need a different compiler than for C; use g++ on Linux. Example:
  
g++ -Wall -o hello hello.cc

• The .cc extension is a convention (just like .c for C), but less universal (also common: .cpp, .cxx, .C, …)

• Uses the C preprocessor (no change there)

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

- Operator $\ll$ takes a “ostream” and (various things) and outputs it; returns the stream, which is why this works:
  
  ```cpp
  std::cout $\ll$ 3 $\ll$ "hi" $\ll$ f(x) $\ll$ '\n';
  ```
  
  - Easier and safer than printf (type safe)

- Operator $\gg$ takes "istream" and (various things) and inputs into it
  
  ```cpp
  int x; std::cin $\gg$ x;
  ```
  
  - Easier and safer than scanf. Do not use pointers –
Can “think of” `>>` and `<<` as keywords, but they are not:
- Operator overloading redefines them for different pairs of types
  - In C and core C++ they mean “left-shift” and “right-shift” (of bits); undefined for non-numeric types
- Lack of address-of for input (`cin >> x`) done with call-by-reference (coming soon)
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 thespace { ... 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
• Example, the whole C++ standard library is in namespace std
• To use a function/variable/etc. in another namespace, do
  
  thespace::some_fun()  (not . like in Java)
Using

• To avoid having to always write namespaces and :: use a *using declaration*

• Example:

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

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

Not quite like Java:
- access-modifier (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; all objects are dynamically allocated on the heap)
- 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, except this is a pointer)

```
    Thing t(10000);
```
- Java: `new Thing(...)` calls constructor, returns heap-allocated pointer
- C: Use `malloc` and then initialized, must free exactly once later, untyped pointers
- C++: Like Java, `new Thing(...)` , but can also do `new int(42)` . Like C must deallocate, but must use `delete` instead of `free`. (never mix malloc/free with new/delete!)
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 – `delete` must be told when it is deleting an array (otherwise it tries to delete a single element)
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 also works in C++; but can also use a “reference parameter” (& character before var name)
• Function definition: \texttt{void f(int& x)} \{x = x+1;\}
• Caller writes: \texttt{f(y)}
• But it’s as though the caller wrote \texttt{f(&y)} and every occurrence of \texttt{x} in the function really said \texttt{*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 the copy-constructor 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
- Copy constructor vs. assignment
  - Copy constructor *initializes* a new bag of bits (new variable or parameter)
  - Assignment (=) *replaces* an existing value with a new one – may need to clean up old state (free heap data?)
**const**

- **const** can appear in many places in C++ code
  - Basically means “doesn’t change” or “won’t change”, but there are subtleties
- Examples:
  ```cpp
  const int default_length = 125; // don’t use #define
  void examine (const thing &t); // won’t change t
  ```
- “const correctness” is important in real C++ code
  - Learn it if you do any non-trivial C++
Still to come

• So far we have classes and objects (class instances)
  – Enough for many interesting types, particularly small concrete types like strings, complex, date, time, etc
• For full object-oriented programming we still need (and have) subclassing, inheritance, and related things
  – Many similarities with Java, but more options and different defaults