CSE 333
Lecture 9 - intro to C++

Steve Gribble
Department of Computer Science & Engineering
University of Washington
Today’s goals

An introduction to C++
- some shortcomings of C that C++ addresses
- give you my perspective on how to learn C++
- kick the tires and write some code
We had to work hard to mimic encapsulation, abstraction

- **encapsulation**: hiding implementation details
  - used header file conventions and the “static” specifier to separate private functions from public functions
  - cast structures to (void *) to hide implementation-specific details

- **abstraction**: associating behavior with encapsulated state
  - the functions that operate on a LinkedList were not really tied to the linked list structure
  - we passed a linked list to a function, rather than invoking a method on a linked list instance
C++

A major addition is its support for object-orientedness!

- classes
  - public, private, and protected **methods** and **instance variables**
  - (multiple!) inheritance

- polymorphism
  - **static polymorphism**: multiple functions or methods with the same name, but different argument types
  - **dynamic polymorphism**: derived classes can overload of methods of parents, and methods will be dispatched correctly
We had to emulate generic data structures

- customer passes a (void *) as a payload to a linked list

- customer had to pass in function pointers so that the linked list could operate on payloads correctly
  - comparisons, deallocation, pickling up state, etc.
C++

Supports **templates** to facilitate generic data types!

- to declare that \( x \) is a vector of ints:
  
  ‣ \( \text{vector<int> } x; \)

- to declare that \( x \) is a vector of floats:
  
  ‣ \( \text{vector<float> } x; \)

- to declare that \( x \) is a vector of (vectors of floats):
  
  ‣ \( \text{vector<vector<float>> } x; \)
We had to be careful about namespace collisions

- C distinguishes between external and internal linkage
  - use “static” to prevent a name from being visible outside a module
  - otherwise, a name is global -- visible everywhere

- we used naming conventions to help avoid collisions in the global namespace
  - LLIteratorNext, HTIteratorNext, etc.
C++

Permits a module to define its own namespace!
- the linked list module could define an “LL” namespace
- the hashtable module could define an “HT” namespace
- both modules could define an Iterator class
  ‣ one would be globally named `LL::Iterator`
  ‣ the other would be globally named `HT::Iterator`
C

C does not provide any standard data structures

- we had to implement our own linked list and hash table
- as a C programmer, you often re-invent the wheel badly
  ‣ maybe if you’re clever you’ll use somebody else’s libraries
  ‣ but, C’s lack of abstraction, encapsulation, and generics means you’ll probably have to tweak them, or tweak your code to use them
C++

The C++ standard library is rich!

- **generic containers**: bitset, queue, list, associative array (including hash table), queue, set, stack, and vector
  - and, iterators for most of these

- **a string class**: hides the implementation of strings

- **streams**: allows you to stream data to and from objects, consoles, files, strings, and so on

- and more...
Error handling is a pain

- have to define error codes and return them

- customers have to understand error code conventions, and need to constantly test return values

- if \texttt{a()} calls \texttt{b()} calls \texttt{c()}

  \textbullet{} \texttt{a} depends on \texttt{b} to propagate an error in \texttt{c} back to it
C++

Supports exceptions!

- try / throw / catch

if used with discipline, can simplify error processing

- but, if used carelessly, can complicate memory management

- consider: a() calls b() calls c()

  ‣ if c() throws an exception that b() doesn’t catch, you might not get a chance to clean up resources allocated inside b()
Some tasks still hurt in C++

Memory management

- C++ has no garbage collector
  - you have to manage memory allocation and deallocation, and track ownership of memory
  - it’s still possible to have leaks, double frees, and so on
- but, there are some things that help
  - “smart pointers”
    - classes that encapsulate pointers and track reference counts
    - deallocate memory when the reference count goes to zero
Some tasks still hurt in C++

C++ doesn’t guarantee type or memory safety

- You can still...
  - forcibly cast pointers between incompatible types
  - walk off the end of an array and smash the stack
  - have dangling pointers
  - conjure up a pointer to an address of your choosing
C++ has many, many features.

Operator overloading
- your class can define methods for handling “+”, “->”, etc!

Object constructors, destructors
- particularly handy for stack-allocated objects

Reference types
- truly pass-by-reference instead of pass-by-value

Advanced OO
- multiple inheritance, virtual base classes, dynamic dispatch
How to think about C++

set of styles and ways to use C++

style guides

good styles and robust engineering practices

set of styles and ways to use C
Or...

in the hands of a disciplined programmer, C++ is a powerful weapon

but, if you’re not so disciplined about how you use C++...
Hello, world!

#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}

Looks simple enough...

- compile with g++ instead of gcc:
  - g++ -Wall -std=gnu++0x -o helloworld helloworld.cc

- let’s walk through the program step by step
Hello, world!

### Includes

```cpp
#include <iostream>
#include <cstdlib>
```

```cpp
t void main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

### Explanation

- **iostream.h** is part of the C++ standard library

- **Note:** you don’t need to include the “.h” when you include standard library headers
  - but you do for local headers (e.g., `#include "ll.h"`)

- **iostream** declares stream object instances, including `std::cin`, `std::cout`, `std::cerr`, in the “std” namespace
Hello, world!

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

cstdlib is the C standard library’s stdlib.h header

- (nearly) all C standard library functions are available to you

  ‣ for header foo.h, you should #include <cfoo>

- we need it for EXIT_SUCCESS, as usual
Hello, world!

```
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

std::cout is the “cout” object instance declared by iostream.h, living within the “std” namespace

- std::cout is an object of class ostream
- used to format and write output to the console
Hello, world!

C++ distinguishes between objects and primitive types

- primitive types include all the familiar ones from C
  - char, short, unsigned long, float, double, long double, etc.
  - and, C++ defines “bool” as a primitive type (woohoo!)
Hello, world!

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

“<<” is an operator defined by the C++ language
- it’s defined by C as well; in C/C++, it bitshifts integers
- but, C++ allows **classes** to overload operators
  ‣ the ostream class overloads “<<”
  ‣ i.e., it defines methods that are invoked when an ostream is the LHS of the << operator
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}

ostream has many different methods to handle `<<`
- the methods differ in the type of the RHS of `<<`
- if you do `std::cout << "foo";`
  ‣ C++ invokes cout’s method to handle `"<<"` with RHS “char *”
Hello, world!

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!" << std::endl;
    return EXIT_SUCCESS;
}
```

the ostream class’s methods that handle “<<” return (a reference to) themselves

- so, when (std::cout << "Hello, World!") is evaluated:
  - a method of the std::cout object is invoked
  - it buffers the string “Hello, World!” for the console
  - and, it returns (a reference to) std::cout
Hello, world!

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv) {
    std::cout << "Hello, World!"
    return EXIT_SUCCESS;
}
```

next, a method on `std::cout` to handle `"<<"` is invoked
- this time, the RHS is `std::endl`
- turns out this is a pointer to a “manipulator” function
  - this manipulator function writes newline to the ostream it is invoked on, and then flushes the ostream’s buffer
  - so, something is printed on the console at this point
Wow...

You should be surprised and scared at this point

- C++ makes it easy to hide a significant amount of complexity
  
  ‣ it’s powerful, but really dangerous

  ‣ once you mix together templates, operator overloading, method overloading, generics, and multiple inheritance, and it gets really hard to know what’s actually happening!
Refining it a bit...

C++'s standard library has a `std::string` class!

- include the string.h header to use it
Refining it a bit...

The “using” keyword introduces part of a namespace, or an entire namespace, into the current region

- using namespace std; -- imports all names from std::

- using std::cout; -- imports only std::cout
Refining it a bit...

```cpp
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

int main(int argc, char **argv) {
    string hello = "Hello, World!");
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```

We’re instantiating a `std::string` object on the stack
- passing the C string “Hello, World!” to its constructor method
  - `hello` is deallocated (and its destructor invoked) when main returns
Refining it a bit...

The C++ string library overloads the `<<` operator as well

- defines a function (not an object method) that is invoked when the LHS is an ostream and the RHS is a std::string

  http://www.cplusplus.com/reference/string/operator<</
Refining it a bit...

```cpp
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

int main(int argc, char **argv) {
    string hello("Hello, World!");
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```

Note the side-effect of using namespace std;

- can now refer to std::string by string, std::cout by cout, and std::endl by endl
string concatenation

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {
    string hello("Hello");
    hello += " there";
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```

The string class overloads the “+” operator
- creates and returns a new string that is the concatenation of LHS and RHS
string assignment

```cpp
class string { // string class  

public:  
    string();  
    string(const string& x);  
    string(const char* s);  
    string(int);  
    void swap(string&);  
    string& operator=(const string&);  
    string& operator+=const string&);  
};

#include <iostream>  
#include <cstdlib>  
using namespace std;

int main(int argc, char **argv) {
    string hello("Hello");
    hello += " there";
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```

The string class overloads the “=” operator

- copies the RHS and replaces the string’s contents with it

  - so, the full statement (a) “+” creates a string that is the concatenation of hello’s current contents and “ there”, and (b) “=” creates a copy of the concatenation to store in hello
stream manipulators

```cpp
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

int main(int argc, char **argv) {
    cout << "Hi! " << setw(4) << 5 << " " << 5 << "\n;  
    cout << hex << 16 << " " << 13 << "\n;  
    cout << dec << 16 << " " << 13 << "\n;  
    return EXIT_SUCCESS;
}
```

iomanip.h defines a set of stream manipulator functions
- pass them to a stream to affect formatting
stream manipulators

```cpp
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

int main(int argc, char **argv) {
    cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
    cout << hex << 16 << " " << 13 << endl;
    cout << dec << 16 << " " << 13 << endl;
    return EXIT_SUCCESS;
}
```

setw(x) sets the width of the next field to x

- only affects the next thing sent to the output stream
stream manipulators

```cpp
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace std;

int main(int argc, char **argv) {
    cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
    cout << hex << 16 << " " << 13 << endl;
    cout << dec << 16 << " " << 13 << endl;
    return EXIT_SUCCESS;
}
```

- `hex` sets the stream to output integers in hexadecimal
  - stays in effect until you set the stream to some other base
- `hex`, `dec`, `oct` are your choices
You can still use printf, though

```c
#include <cstdio>
#include <cstdlib>

int main(int argc, char **argv) {
    printf("hello from C\n");
    return EXIT_SUCCESS;
}
```

C is (roughly) a subset of C++
Reading

std::cin is an object instance of class istream
- supports the >> operator for “extraction”
- cin also has a getline() method

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {
    int num;
    cout << "Type a number: ";
    cin >> num;
    cout << "You typed: " << num << endl;
    return EXIT_SUCCESS;
}
```
Exercise 1

Write a C++ program that:

- uses streams to:
  ‣ prompts the user to type in 5 floats
  ‣ prints them out in opposite order
  ‣ with 4 digits of precision
See you on Friday!