CSE 333

Lecture 13 - intro to C++

John Zahorjan

Department of Computer Science & Engineering

University of Washington

Today’s goals

An introduction to C++

- some shortcomings of C that C++ addresses
- give you a perspective on how to learn C++
- kick the tires and write some code

Advice: read related sections in the *C++ Primer*. It’s hard to learn the “why is it done like this” from reference docs

C

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 classes & objects!

- 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 (overloading)
    - Works for all functions, not just class members
  - **dynamic (subtype) polymorphism**: derived classes can override methods of parents, and methods will be dispatched correctly

C

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!

- Parametric polymorphism - same idea as Java generics, but different in details - particularly implementation

- to declare that **x** is a vector of ints:
  - `vector<int> x;`
- to declare that x is a vector of floats:
  - `vector<float> x;`
- to declare that x is a vector of (vectors of floats):
  - `vector<vector<float>> x;`

C

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 source file (as close as C gets to “private”)
  - otherwise, a name is global -- visible everywhere
- we used naming conventions to help avoid collisions in the global namespace
  - **LL**IteratorNext, **HT**IteratorNext, 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`

Classes also allow duplicate names without collisions

- Namespaces isolate names in collections of classes and other "global" things (somewhat like Java packages)

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), deque, 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...

C

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 **a**( ) calls **b**( ) calls **c**( )
  - **a** depends on **b** to propagate an error in **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( )

But much C++ code still needs to work with C & old C++ libraries, so still uses return codes, exit( ), etc.

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 (or heap)
  - 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

Hello, world!

helloworld.cc

```
#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 -g -std=c++11 -o helloworld helloworld.cc`
- let's walk through the program step by step

Hello, world!

helloworld.cc

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

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

iostream is part of the C++ standard library

- note you don't include a ".h" when you include C++ 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!

helloworld.cc

```
#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 standard header `<foo.h>`, you should `#include <cfoo>`
- we need it for EXIT_SUCCESS, as usual

Hello, world!

helloworld.cc

```
#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, living within the “std” namespace (C++’s name for stdout)

- std::cout is an object of class ostream
  - http://www.cplusplus.com/reference/iostream/ostream/
- used to format and write output to the console
- the entire standard library is in namespace std

Hello, world!

helloworld.cc

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

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

C++ distinguishes between objects and primitive types

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

Hello, world!

helloworld.cc

```
#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 define the meanings of operators applied to their instances
  - the ostream class overloads "<<"
  - i.e., it defines methods that are invoked when an ostream is the LHS of the << operator

Hello, world!

helloworld.cc

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

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

<< is a binary operator

- the two operands are written to its left and right
- 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!

helloworld.cc

```
#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!

helloworld.cc

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

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

helloworld.cc

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

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

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, it gets really hard to know what's actually happening!

Refining it a bit...

helloworld2.cc

```
#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;
}
```

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

- include the <string> header to use it
- http://www.cplusplus.com/reference/string/

Refining it a bit...

helloworld2.cc

```
#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;
}
```

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…

helloworld2.cc

```
#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...

helloworld2.cc

```
#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;
}
```

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...

helloworld2.cc

```
#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

concat.cc

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

using namespace std;

int main(int argc, char **argv) {
  string hello("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

concat.cc

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

using namespace std;

int main(int argc, char **argv) {
  string hello("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 (i) “+” creates a string that is the concatenation of hello’s current contents and “ there”, and (ii) “=” creates a copy of the concatenation to store in hello. Without the syntactic sugar it is:
    `hello.operator=(hello.operator+(" there"));`

stream manipulators

helloworld3.cc

```
#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;
}
```

iomanip defines a set of stream manipulator functions

- pass them to a stream to affect formatting
  - http://www.cplusplus.com/reference/iostream/manipulators/

stream manipulators

helloworld3.cc

```
#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

helloworld3.cc

```
#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

helloworld4.cc

```
#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++

- Can mix C and C++ idioms if needed to work with existing code, but avoid mixing if you can - use C++(11)

Reading

helloworld5.cc

```
#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;
}
```

std::cin is an object instance of class istream

- supports the >> operator for “extraction”
- cin also has a getline( ) method

Suggested exercise

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