C++ Intro

CSE 333 Winter 2021

Instructor: John Zahorjan

Teaching Assistants:

Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Today's Goals

- An introduction to C++
  - Some comparisons to C and shortcomings that C++ addresses
  - Give you a perspective on how to learn C++
  - Kick the tires and look at some code

- **Advice:**
  - C++ is much bigger and more complicated than C
  - Web searches for help on some particular problem you're having may not be as successful
  - In any case, it would be worth reading some prose discussion of each C++ topic (a textbook, say, or an article you trust)

C++

- C is roughly a subset of C++
  - Most C program can be compiled with a C++ compiler and mean the same things they mean in C
- That means these basic concepts are preserved:
  - global / local / heal allocated variables
  - pointers
  - assignment is (by default) memory copy
  - call by value
  - a single (default) global name space for functions
  - declare / define distinction
  - A “you’re the boss” attitude – if it can be compiled, the compiler is likely to compile it

C++

- C++ has evolved considerably over time
- It's hard to get rid of language features..
  - Sometimes the language is a little more rough edged than it would be if we started over and designed it today
- Sometimes it's hard to figure out correct syntax
- Sometimes it's hard to know for sure what a statement means
- C++ does much more sophisticated compile time code analysis than C
  - Used mainly to make it more expressive

C++

- A major addition is support for classes and 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++ is MUCH MORE than the addition of classes, though!

Namespaces - C

- We had to be careful about namespace collisions
  - We used naming conventions to help avoid collisions in the global namespace
    - *e.g.* **`LL`**`IteratorNext` vs. **`HT`**`IteratorNext`, etc.

Namespaces - C++

- Permits creation of namespaces
  - The linked list module could define an "`LL`" namespace while the hash table module could define an "`HT`" namespace
  - Both modules could define a class with (local) name Iterator
    - One would be globally named `LL::Iterator`
    - The other would be globally named `HT::Iterator`
- Classes also allow duplicate names without collisions
  - Namespaces group and isolate names in collections of classes and other "global" things (somewhat like Java packages)
    - Entire C++ standard library is in a namespace `std` (more later…)

Polymorphism - C

- Nope

Polymorphism – C++

- Yep
  - Person::update(string s); *and*
    Person::update(int x);

- *In fact, C++ views most everything you write as a request to invoke some functionality, and then allows the programmer to (re)define that functionality*
  - The language is "exporting" control over the meaning of operators, say, to the programmer
  - A very general mechanism is used for the programmer to express the meaning: code!

Generics - C

- We had to emulate generic data structures
  - Generic linked list using `void*` payload
  - Pass function pointers to generalize different "methods" for data structures
    - Comparisons, deallocation, pickling up state, etc.

Generics - 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 strings: `vector<string> x;`
  - To declare that x is a vector of (vectors of floats): `vector<vector<float>> x;`

- We write code that, in essence, generates code...

Standard Library - C

- C doesn't provide any standard data structures
  - We had to implement our own linked list and hash table
  - As a C programmer, you often reinvent the wheel
    - 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 end up tinkering with them or tweak your code to use them

Standard Library - C++

- The C++ standard library is huge!
  - **Generic containers:** bitset, queue, list, associative array (including hash table), deque, set, stack, and vector
    - And iterators for most of these
    - And algorithms for most of these...
  - **A `string` class:** yeah!
  - **Streams:** allows you to stream data to and from objects, consoles, files, strings, and so on
  - And more…

- *Many of the features that have been introduced into C++ over the years have to do with writing efficient libraries*

Error Handling - C

- There is no language support, only convention
- Convention:
  - Define and return error codes
  - Customers have to understand error code conventions and need to constantly test return values
  - *e.g.* if `a()` calls `b()`, and `b()` calls `c()`
    - `a` depends on `b` to propagate an error in `c` back to it

Error Handling - C++

- Supports exceptions
  - `try` / `throw` / `catch`
  - Can simplify error processing, but...
  - There is an unfortunate interaction with memory management
    - Consider: `a()` calls `b()`, which calls `c()`
      - If `c()` throws an exception that `b()` doesn't catch, `b()` might not get a chance to free resources it allocated → memory leak

- C++ code often needs to work with C libraries that use return codes
  - Including library routines making system calls (e.g., I/O)
    - Some of which still use errno

C++ Hilarity

```
void sub()
{
  std::cout << "sub() here\n";
}

int main()
{
  try
    {
      throw sub;
    }
  catch (void (*proc)())
    {
      proc();
    }
  return 0;
}
```

I hope you'll never actually do this!

```
[attu3] ~/tmp> g++ -std=c++17 -g -Wall throw-joke.cc -o throw-joke
[attu3] ~/tmp> ./throw-joke
sub() here
```

C++ Additional Hilarity

```
void sub()
{
  std::cout << "sub() here\n";
}

int main() noexcept
{
  try
    {
      throw sub;
    }
  catch (void (*proc)())
    {
      proc();
    }
  return 0;
}
```

Yikes!

```
[attu3] ~/tmp> g++ -std=c++17 -g -Wall throw-joke.cc -o throw-joke
[attu3] ~/tmp> ./throw-joke
sub() here
```

C++ is C's Crazed Offspring

- C++ shares many of the attitudes of its parent
  - Execution performance should be as good as, or better, than what a team of assembler prorammers could produce
- Memory management
  - C++ has no garbage collector
    - If you use new/malloc, you're responsible for delete/free
  - But some other features help
    - Classes let you build "smart pointers" that can do reference counted garbage collection
  - Awesome, but it will take a bit to see why:
    - C++ guarantees that the constructor is called when an object is created
    - It also guarantees that the destructor is called when it is destroyed
      - Think about that property and the fact that you can stack allocate objects

C++ is Still a Crazy Mix of Execution in the Language and Execution in the Hardware

- 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 memory
    - Have dangling pointers (pointers pointing to memory that has been freed)
    - Create a pointer to an arbitrary address
    - Declare things "private" and then get around it
    - (Sometimes) declare things const and then find a way to modify them

C++ Has Many, Many Features

- Operator overloading
  - Your class can define methods for handling "+", "->", etc.
    - You can make '+' mean subtract!
- Object constructors, destructors
  - Particularly handy for stack-allocated objects
- Reference types
  - Truly pass-by-reference instead of always pass-by-value
- Advanced Objects
  - Multiple inheritance, virtual base classes, dynamic dispatch
- (Almost) All the features have some specified meaning, so that compilers can implement them
  - Sometimes the rules are so complicated you can't apply them

Moving Toward Understanding C++

- void sub(const myStruct *pStruct);
  - You're thinking, *"Great, C++ will guarantee for me that my structure isn't changed if I pass it to sub"*
  - C++ is thinking, *"Great, the programmer is telling me I can assume that structure isn't changed when they call sub"*
  - (Note: You might reasonably be thinking "what does const myStruct *" mean? That pStruct can't change or that *pStruct can't change, or both?)
- ...

```
myStructInstance.nUnits = 1;
sub(&myStructInstance);
totalUnits = totalUnits + myStructInstance.nUnits;
```

Hello World in C

```
#include <stdio.h>     // for printf()
#include <stdlib.h>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  printf("Hello, World!\n");
  return EXIT_SUCCESS;
}
```

- Compile with `gcc`:

```
gcc -Wall -g -std=c17 -o hello helloworld.c
```

- You should be able to describe in detail everything in this code

Hello World in C++

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`
  - Use `.cc` files instead of `.c`

```
g++ -Wall -g -std=c++17 -o helloworld helloworld.cc
```

  - Let's walk through the program step-by-step to highlight some differences

Hello World in C++

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 write "`.h`" when you include C++ standard library headers
    - But you *do* for local headers (*e.g.* `#include "ll.h"`)
  - `iostream` declares stream *object* instances in the "`std`" namespace
    - *e.g.* `std::cin`, `std::cout`, `std::cerr`

Hello World in C++

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`
  - We include it here for `EXIT_SUCCESS`, as usual
  - Nearly all C standard library functions are available to you
    - For C header `some.h`, you should `#include <csome>`

Hello World in C++

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 in the "`std`" namespace (declared by `iostream`)
  - C++'s name for `stdout`
  - `std:cout` is an object of class `ostream`
    - http://www.cplusplus.com/reference/ostream/ostream/
- The entire standard library is in the namespace `std`

Hello World in C++

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
  - These include the familiar ones from C:
    `char`, `short`, `int`, `long`, `float`, `double`, etc.
  - C++ also defines `bool` as a primitive type
    - But bool and int values silently convert types for compatiblity with C

Hello World in C++

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
  - *Defined in C as well: it bit-shifts integers in C (and C++)*
  - C++ allows classes and functions to overload operators!
    - Here, the `ostream` class overloads "<<"

Operators in C++ (preview)

- *In C++, everything is a function call* (only kind of true)
- In C:

  LinkedList_Append(&list, payload)

- In Java:

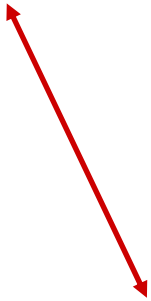
  list.append(payload)

- In C++:

  list.append(payload) // append is a binary function

  or

  list + payload // "+" is the name of a binary function



Hello World in C++

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
  - *Defined in C as well: it bit-shifts integers in C (and C++)*
  - Here, the `ostream` class defines the function "<<" when a char* is given as the (second) input

Hello World in C++

helloworld.cc

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

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

- The `ostream` class' member functions that handle `<<` return a reference to themselves
  - When `std::cout << "Hello, World!";` is evaluated:
    - A member function of the `std::cout` object is invoked
    - It buffers the string `"Hello, World!"` to stdout
    - It returns (a reference to) `std::cout`
      - "method chaining"

Hello World in C++

helloworld.cc

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

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

- Next, another member function on `std::cout` is invoked to handle << with RHS `std::endl`
  - `std::endl` is a pointer to a "manipulator" function
    - This manipulator function writes newline (`'\n'`) to the `ostream` it is invoked on and then flushes the `ostream`'s buffer
    - This *enforces* that something is printed to the console at this point

With Objects

helloworld2.cc

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

int main(int argc, char** argv) {
  std::string hello("Hello, World!");
  std::cout << hello << std::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/

With Objects

helloworld2.cc

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

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

- Here we are instantiating a `std::string` object *on the stack* (an ordinary local variable)
  - Passing the C string `"Hello, World!"` to its constructor
  - Don't have to "new" to create an object
- *hello is deallocated (and its destructor invoked) when main returns*

With Objects

helloworld2.cc

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

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

- The C++ string library also overloads the << operator
  - Defines a function that is invoked when the LHS is `ostream` and the RHS is `std::string`
    - http://www.cplusplus.com/reference/string/string/operator<</
- We'll look at this in detail later…

using namespace std;

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 a namespace (or part of) into the current region
  - `using namespace std;` imports all names from `std::`
  - `using std::cout;` imports *only* `std::cout` (used as `cout`)

using namespace std;

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 can now refer to `std::string` as `string`, `std::cout` as `cout`, and `std::endl` as `endl`
  - Google style guide says never use `using namespace`, only `using` for individual items
  - `using namespace std;` is used, a lot
  - Eschew using it…

String Concatenation

concat.cc

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

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

- The string class overloads the "+" operator with argument of type char*
- Apparently just like Java!
  - The effect is just what you expect
  - Except some much more complicated things are actually going on…

String Assignment

concat.cc

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

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

- The string class overloads the "=" operator
- The effect is just like Java!
  - What is happening is more complicated…

Alternate Syntax

concat.cc

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

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

```
hello.operator=(hello.operator+(", World!"));
```

Stream Manipulators

manip.cc

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

int main(int argc, char** argv) {
  std::cout << "Hi! " << std::setw(4) << 5
            << " " << 5 << std::endl;
  cout << std:: hex << 16 << " " << 13 << std:: endl;
  cout << std:: dec << 16 << " " << 13 << std:: 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/iomanip/
    - http://www.cplusplus.com/reference/ios/

Stream Manipulators

manip.cc

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

int main(int argc, char** argv) {
  std::cout << "Hi! " << std::setw(4) << 5
            << " " << 5 << std::endl;
  cout << std:: hex << 16 << " " << 13 << std:: endl;
  cout << std:: dec << 16 << " " << 13 << std:: 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 (*i.e.* it is not persistent)

Stream Manipulators

manip.cc

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

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

- `hex`, `dec`, and `oct` set the numerical base for *integer* output to the stream
  - Stays in effect until you set the stream to another base (*i.e.,* it is persistent)

C and C++

helloworld3.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++
  - You can still use `printf` – but bad style in ordinary C++ code
  - Can mix C and C++ idioms if needed to work with existing code, but avoid mixing if you can
    - Use C++(17)

Reading Input

echonum.cc

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

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

- `std::cin` is an object instance of class `istream`
  - Supports the >> operator for "extraction"
    - Can be used in conditionals – (`std::cin>>num`) is `true` if successful
      - How is that possible?
  - Has a `getline()` method and methods to detect and clear errors