Introduction to C++

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi
Nathan Lipiarski
Yibo Cao

Farrell Fileas
Renshu Gu
Yifan Bai

Lukas Joswiak
Travis McGaha
Yifan Xu

Poll Everywhere

pollev.com/cse333

About how long did Exercise 6 take?

A. 0-1 Hours

B. 1-2 Hours

C. 2-3 Hours

D. 3-4 Hours

E. 4+ Hours

F. I didn’t finish / I prefer not to say

Administrivia

- Exercise 7 released today, due Wednesday

- Homework 2 due next Thursday (10/24)
  - File system crawler, indexer, and search engine with C-style inheritance!
  - Remember to place a copy of libhw1.a in the hw1/ directory
    - Either yours (which gets generated there) or ours (copy from hw1/solution_binaries)
  - Demo: Use Ctrl+D to exit, test on your own small directory

Today's Goals

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

- **Advice:** Read related sections in the *C++ Primer*
  - It's hard to learn the "why is it done this way" from reference docs, and even harder to learn from random StackOverflow/GitHub/etc on the web
  - Lectures and examples will introduce the main ideas, but aren't everything you'll ~~want~~ need to understand

C: Encapsulation, Abstraction, OOP

- Header file conventions and the `static` specifier to separate "private" functions/definitions/constants from "public"

- Forward-declared `structs` and opaque pointers to hide implementation-specific details

- Cannot associate behavior with encapsulated state
  - LinkedList "methods" not really tied to `struct LinkedList`

tl;dr: Implemented primarily via coding conventions

C++: Encapsulation, Abstraction, OOP

- Classes! 🎉 🎉 🎉 Objects! 🎉 🎉 🎉
  - Public, private, and protected access specifiers
  - **Methods** and **instance variables** ("this")
  - (Multiple 😱 !) inheritance

- Polymorphism
  - **Static polymorphism** ("overloading"): multiple functions or methods with the same name but different argument types
    - Works for all functions, not just class members
  - **Dynamic (subtype) polymorphism**: derived classes can override parent's methods, and methods will be dispatched correctly

C: Generics

- Generic linked list / hash table using `void*` payload
  - `LLPayload_t p = (LLPayload_t)256L;  //` 😱

    "let's pretend this number is an address"
- Function pointers to generalize different behaviour for data structures
  - Comparisons, deallocation, pickling up state, etc.

tl;dr: Implemented primarily by disabling type system

C++: Generics

- Templates to facilitate generic data types
  - Parametric polymorphism: same idea as Java generics, but different in details, particularly implementation
  - A vector of ints: `vector<int> x;`
  - A vector of floats: `vector<float> x;`
  - A vector of (vectors of floats): `vector<vector<float>> x;`
- Specialized casts to increase type safety eg dynamic_cast() for safe downcasting

  -
    ```
    LLPayload_t p =
    static_cast<LLPayload_t>(256);   // lol no
    ```
    storing an integer inside a ptr? Still possible. Still inadvisable.

C: Namespaces

- Names are global and visible everywhere
  - Can use `static` to prevent a name from being visible outside a source file (as close as C gets to “private”)

- Naming conventions to avoid collisions in global namespace
  - *e.g.* `LinkedList_Allocate` vs. `HTIterator_Next`, etc.

tl;dr: Implemented primarily via coding conventions

C++: Namespaces

- Explicit namespaces!
  - The linked list module could define an "`LL`" namespace while the hash table module could define an "`HT`" namespace
  - Both modules could define an Iterator class
    - One would be globally named `LL::Iterator` and the other would be globally named `HT::Iterator`

- Classes also allow duplicate names without collisions
  - Classes can also define their own pseudo-namespace, very similar to Java static inner classes

C: Standard Library

- C does not provide any standard data structures
  - We had to implement our own linked list and hash table
- Hopefully you can use somebody else's libraries
  - But C's lack of abstraction, encapsulation, and generics means you'll probably need to tweak them or tweak your code in order to use

tl;dr: YOU implement the data structures you need

C++: Standard Library

- **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
- **Generic algorithms**: sort, filter, remove duplicates, etc.

C: Error Handling

- Define error codes and return them
  - Either directly or via a “global” like `errno`
  - No type-checking: does `1` mean `EXIT_FAILURE` or `true`?

- Customers and implementors need to constantly test return values
  - *e.g.* if `a()` calls `b()`, which calls `c()`
    - `a` depends on `b` to propagate an error in `c` back to it

tl;dr: Mixture of coding conventions and discipline

C++: Error Handling

We have RAII instead

- Supports exceptions!
  - `try` / `throw` / `catch`, but no `finally`
  - If used with discipline, can simplify error processing
  - If used carelessly, can complicate memory management
    - Consider: `a()` calls `b()`, which 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()`

- We will largely avoid in 333
  - You still benefit from having more interpretable errors!

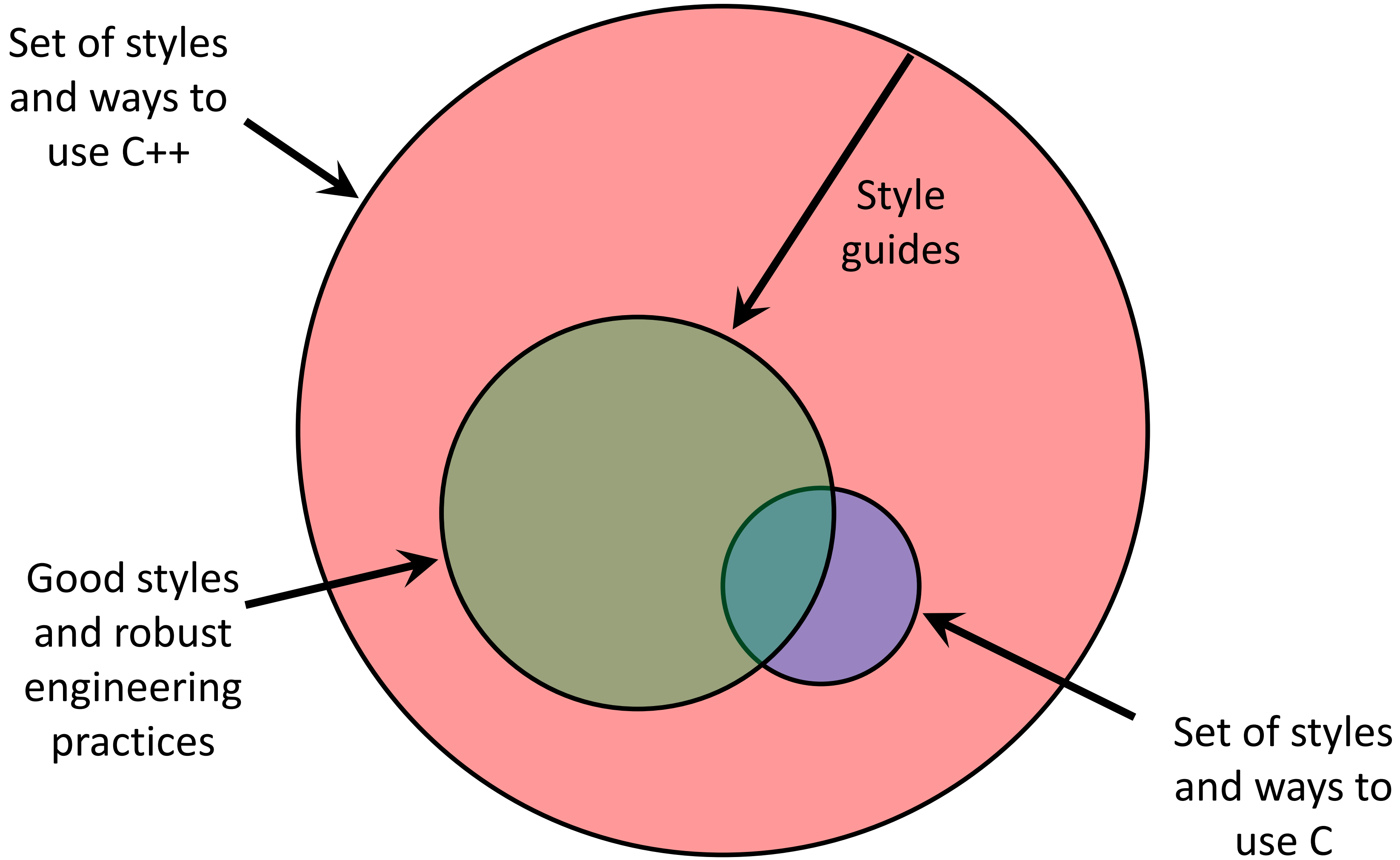
Some Tasks Still Hurt in C++

- Memory management
  - C++ has no garbage collector
    - You have to manage allocation / deallocation and track
    - 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
    - C++'s destructors permit a pattern known as "Resource Allocation Is Initialization" (RAII) (terrible name, nice functionality)
      - Useful for releasing memory, locks, database transactions, and more

Some Tasks Still Hurt in C++

- C++ doesn’t guarantee type or memory safety
  - You can still:
    - Forcibly cast one type to an incompatible type
    - Walk off the end of an array and smash memory
    - Have dangling pointers
    - Conjure up a pointer to an arbitrary address of your choosing

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 tool

But if you're not so disciplined about how you use C++…

Hello World in C

helloworld.c

```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;
}
```

- You never had a chance to write this!
  - Compile with `gcc`:

```
gcc -Wall -g -std=c11 -o helloworld helloworld.c
```

  - Based on what you know now, describe to your neighbor everything that goes on in the execution of this "simple" program
    - Be detailed!

Hello World in C++

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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 to highlight some differences

Hello World in C++

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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

- `cstdlib` is the ***C*** standard library's `stdlib.h`
  - Nearly all C standard libraries are still available
    - For C header `foo.h`, you should `#include <cfoo>`
  - We include it here for `EXIT_SUCCESS`, as usual

Hello World in C++

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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 (woo-hoo!)
    - Use it!

Hello World in C++

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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 instance of class `ostream`
    - http://www.cplusplus.com/reference/ostream/ostream/
  - Used to format and write output to the console
  - The entire standard library is in the namespace `std`

Hello World in C++

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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 and Java as well: usually it bit-shifts integers (in C/C++) Scream
  - C++ allows classes and functions to overload operators! ← emoji here
    - Here, the `ostream` class overloads "<<"
    - *i.e.* it defines different member functions (methods) that are invoked when an `ostream` is the left-hand side of the << operator

ostream& operator<<(string s)

is called as → cout << "hi";

Hello World in C++

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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

overloaded methods!

- `ostream` has many different methods to handle <<
  - The functions differ in the type of the right-hand side (RHS) of <<
  - *e.g.* if you do `std::cout << "foo";` , then C++ invokes `cout`'s function to handle << with RHS `char*`

Hello World in C++

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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!"` for the console
    - And it returns a reference to `std::cout`

Hello World in C++

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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 `std::endl` as its param
  - `std::endl` is a “stream 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

Wow…

helloworld.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

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 everything together (templates, operator overloading, method overloading, generics, multiple inheritance), it can get *really* hard to know what's actually happening!

Extra Exercise #1

- Write a C++ program that uses stream to:
  - Prompt the user to type 5 floats
  - Prints them out in opposite order with 4 digits of precision