CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- Introduction to C++
  - Why C++?
  - Object Oriented Programming
  - Hello, world!

Your Goals

- Compile a C++ program using g++

C v. Scripting

C

- Compiled
- Highly structured, data-typed
- Strings have library processing
- Data structures and libraries
- Good for large complex programs
  - Java, with object-oriented programming, is even better for complex programs

Scripting

- Interpreted
- Esoteric variable access
- Everything is a string
- Easy access to files and program
- Good for quick & interactive programs
  - Do one thing and do it well

C v. Java

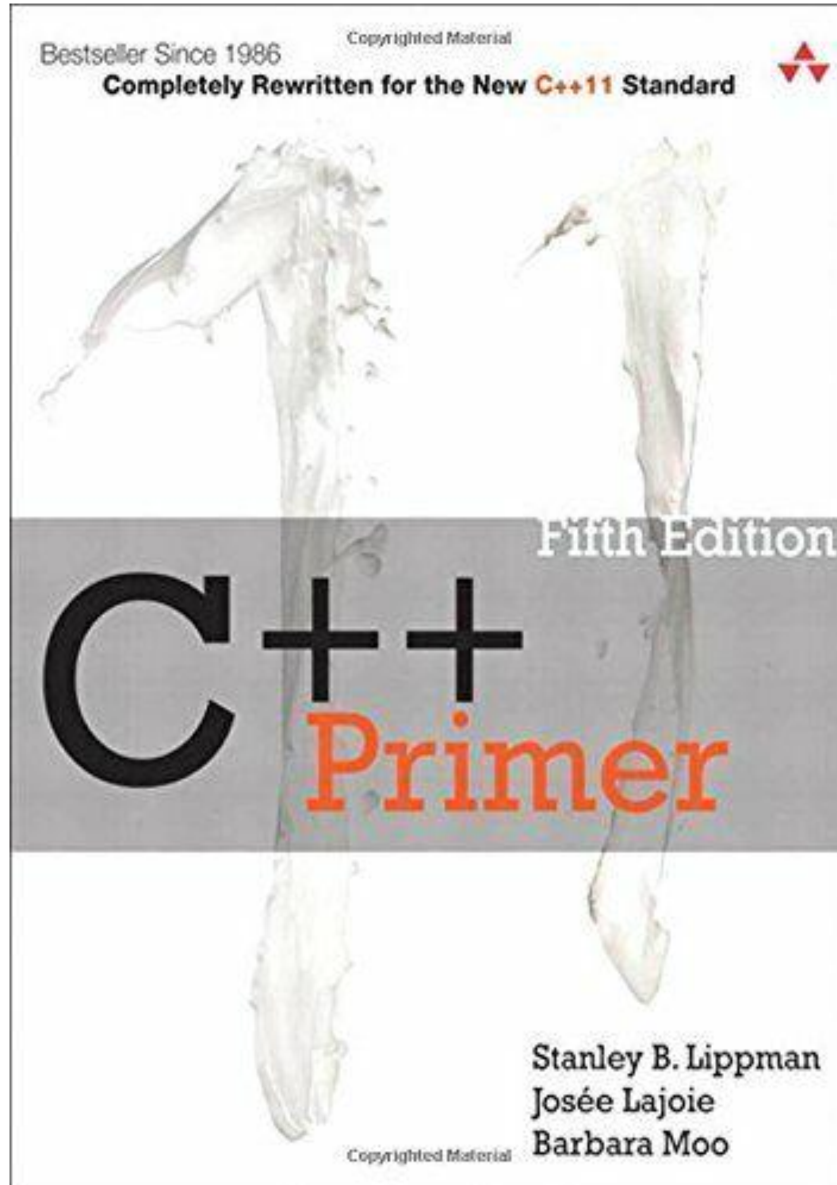
C	Java
• Lower level (closer to assembly)	• Higher level (lots of compilation)
• No guaranteed memory safety	• Safe (sand-boxed in jvm, compiled limits)
• Procedural	• Object Oriented
• Compiled (not interpreted like bash)	• Compiled
• Conditional controls (if, while)	• Conditional controls (if, while)
• Modern syntax (human readable)	• Modern syntax (human readable)
• Small standard library	• Large standard library, huge extended libraries

C v. C++

C	C++
• Lower level (closer to assembly)	• Still user-managed memory / headers
• No guaranteed memory safety	• Still not guaranteed, but tools exist
• Procedural	• Object Oriented
• Compiled (not interpreted like bash)	• Compiled
• Conditional controls (if, while)	• Conditional controls (if, while)
• Modern syntax (human readable)	• Modern syntax (human readable) (??)
• Small standard library	• Standard template library

Why C++?

- Most code bases that use C also use C++
  - Provides modern OOP tools, large libraries
- Understanding C++ can help us understand both C & Java better
- Still used for
  - Embedded programming
  - Systems programming
  - High-performance code
  - GPU Programming
  - C/C++ has more legacy code than any language except COBOL (by some estimates)



C++ References

- Best place to start: **C++ Primer**, Lippman, Lajoie, Moo, 5th ed., Addison-Wesley, 2013
  - Free access through UW libraries
- Every serious C++ programmer should also read: **Effective C++**, Meyers, 3rd ed., Addison-Wesley, 2005
  Best practices for standard C++
- O'Reilly books (such as **Effective Modern C++**)– available through the UW libraries.
- Good online source: Cplusplus: http://www.cplusplus.com/

What is C++

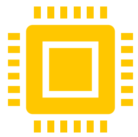
- Developed by Bjarne Stroustrup at Bell Labs
- C with Classes
  - Specifically designed to add OOP features such as classes while maintaining efficiency and flexibility of C
- C++ (C incremented) in 1983, C++11 (Modern C++) in 2011
- C++ is a superset of C
- Dominant language for performance-critical systems such as operating systems, game engines, etc.

What is Object Oriented Programming

- Programming paradigm organized around objects
  - ~1961-67 Simula used in Norway for research on physical modeling
  - ~ 1966 Alan Kay created SmallTalk @ Xerox PARC
- OOP support in a wide range of languages (Lisp, Ada, Fortran 2003, Python Objective-C, R) – but not all of those are solely classified as OOP.

Is C++ Object Oriented? Is Python Object Oriented?

Object Oriented Programming



Encapsulation

Discrete portions of code keep state and implementation private while providing public interfaces

Abstraction

The high-level interface is exposed to users without detailing underlying code.

Inheritance

Classes can be derived from other classes allowing for shared code.

Polymorphism

Subclasses implement methods of superclasses to allow for a consistent interface

OOP ideals in the language

C

- Mimicked with other tools
  - Using header files and static to separate private functions
  - Casting structures to `void*` to hide implementation details
  - Functions receive structs to operate on instead of structs owning their own methods
    - So, structs need to be designed to meet other functional requirements

C++

- Directly implements
  - Classes have public, private, and protected attributes
  - Multiple inheritance
  - Polymorphism – can overload functions or methods with different argument types
  - Derived classes can override parent's methods

Namespaces

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, name is global and visible everywhere
  - We used naming conventions to help avoid collisions in the global namespace
    - *e.g.* `LLIteratorNext` vs. `HTIteratorNext`, etc.

C++

- Permits a module to define its own namespace!
  - 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
    - 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…)

Data structures

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 reinvent the wheel… poorly
    - 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

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
  - **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…

Generic data structures

C

- Can use `void*`
- Can use function pointers to generalize different functions

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;`

Error Handling

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
  - *e.g.* if a() calls b(), which calls c()
    - a depends on b to propagate an error in c back to it

C++

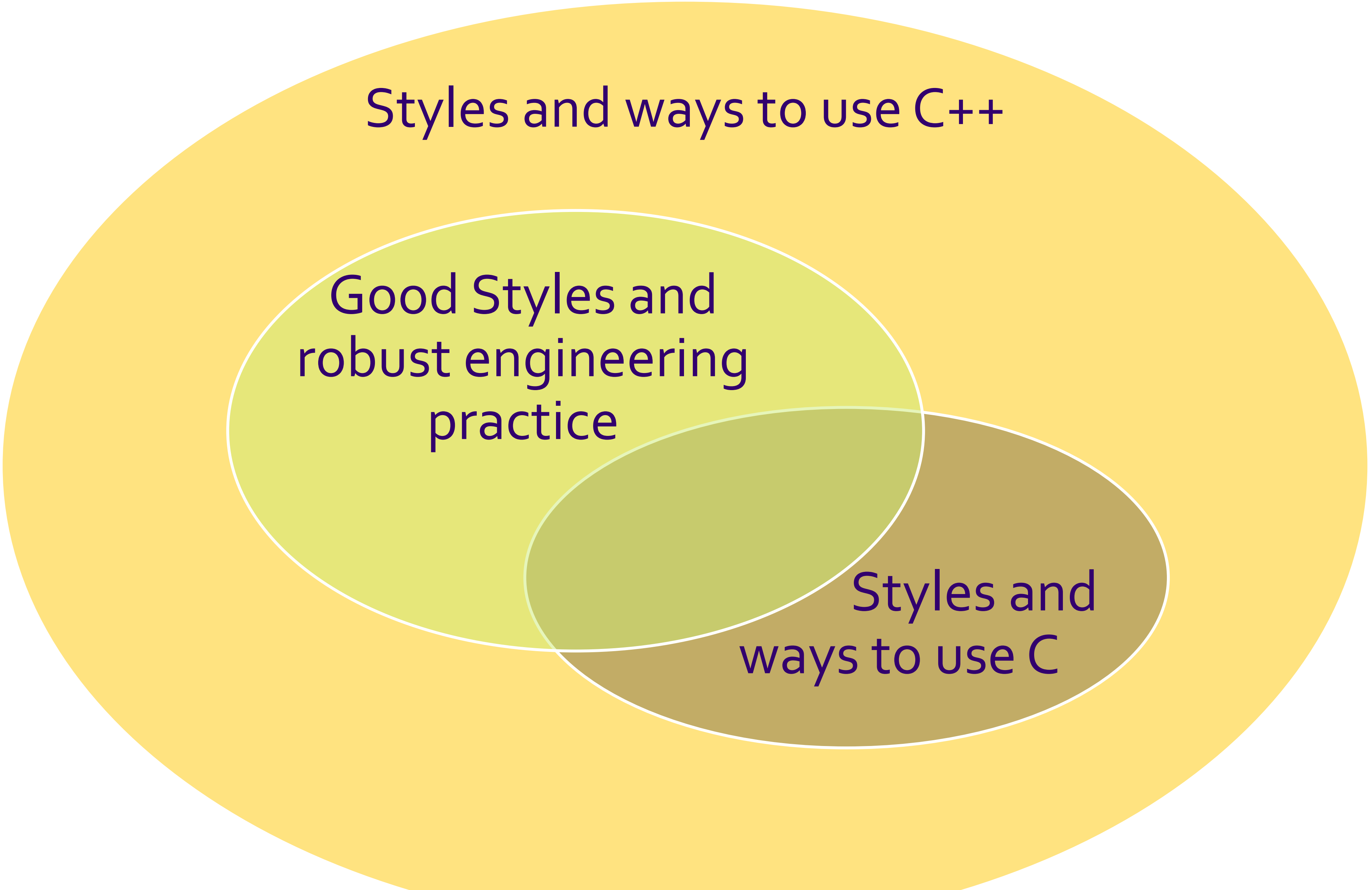
- Error handling is STILL a pain, but now we have exceptions
  - try / throw / catch
  - If used with discipline, can simplify error processing
    - But, if used carelessly, can complicate memory management
    - Consider: a() calls b(), which calls c()
      - If c() throws an exception that b() doesn't catch, you might not get a chance to clean up resources allocated inside b()
  - But much C++ code still needs to work with C & old C++ libraries that are not exception-safe, so still uses return codes, exit(), etc.
    - We won't use (and Google style guide doesn't use either)

C++ has a lot of features

- Operator overloading
  - Your class can define methods for handling “+”, “->”, etc.
- Object constructors, destructors
  - Particularly handy for stack-allocated objects
- Reference types
  - True call-by-reference instead of always call-by-value
- Advanced Objects
  - Multiple inheritance, virtual base classes, dynamic dispatch
- … And its easy to mis-use or poorly-use these features!

C++ still has challenges!

- 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
    - Conjure up a pointer to an arbitrary address of your choosing
- 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
    - C++'s destructors permit a pattern known as "Resource Allocation Is Initialization" (RAII) (terrible name but super useful idea)
      - Useful for releasing memory, locks, database transactions, and more


Styles and ways to use C++
Good Styles and robust engineering practice
Styles and ways to use C

Hello World in C

- Compile at a bash command line with `gcc hello.c`
  - Creates an executable `a.out`
- `gcc -Wall -std=c11 -o hello hello.c`
  - `Wall` turns on all warnings
  - `C11` specifies C11 standard
  - Creates executable hello
- Run: `./a.out` or `./hello`

```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *     gcc -o hello hello.c
 * Run the program:
 *     ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

Hello World in C++

- `g++ -Wall -std=c++17 -o helloworld helloworld.cc`
  - `Wall` turns on all warnings
  - `C++17` specifies C++ standard
  - Creates executable hello
- Run: `./helloworld`

```
#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;
}
```

Hello World in C++: Include

- `#include` style is different
  - No `.h` for c++ lib.
  - `Local.h` for local lib
- `<iostream>` replaces `<stdio.h>`
- `<cstdlib>` is `<stdlib.h>` : we can use it here, and do for `EXIT_SUCCESS`

```
#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;
}
```

Hello World in C++: iostream

- `<iostream>` gives us objects and operators
- `Std::cout` is an object (`cout`) of type `ostream`, declared in a namespace (`std`)
- `<<` is an operator defined by C++. The class `ostream` overloads `<<` to print when it is on the left-hand side of the operator
- Fun fact: << does bit-shifts in C

Hello World in C++: << operator

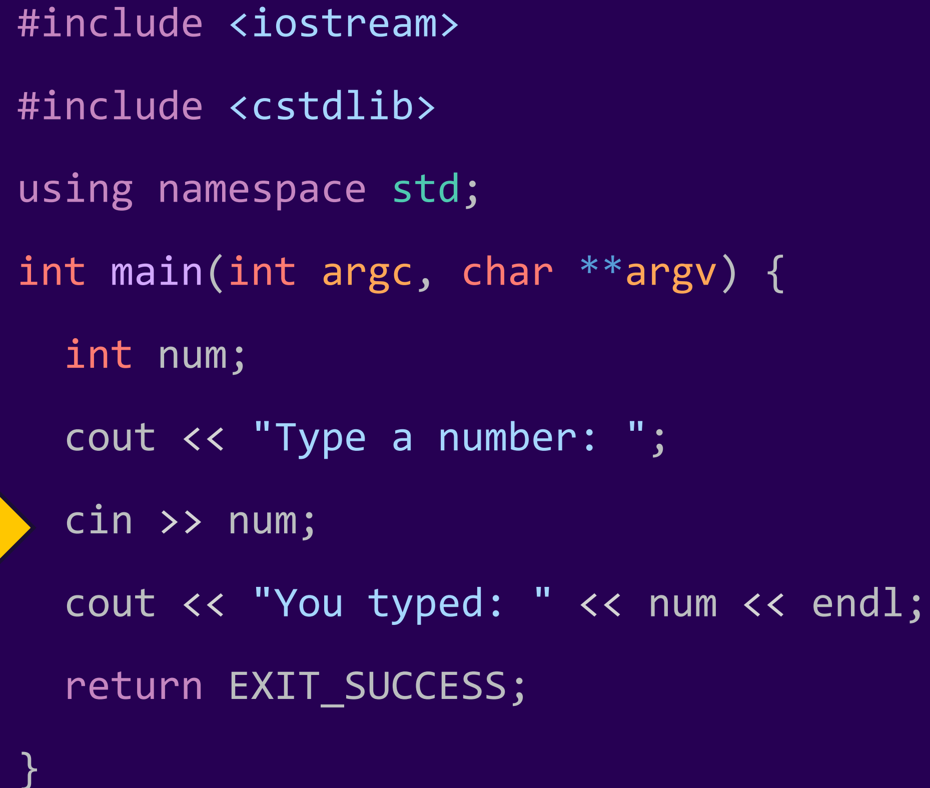
- `<<` is an operator
- Different values can be on the right-hand side of `<<`.
- "Hello, World!" is still a `char*` here.
- `std::endl` is a pointer to function that writes a newline ("`\n`") and flushes `ostream`'s buffer.
  - Forces a printout

```
#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;
}
```

Hello World in C++: >> operator

- `>>` is an operator
- `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
- Has a `getline()` method and methods to detect and clear errors



C in C++

C is (roughly) a subset of C++

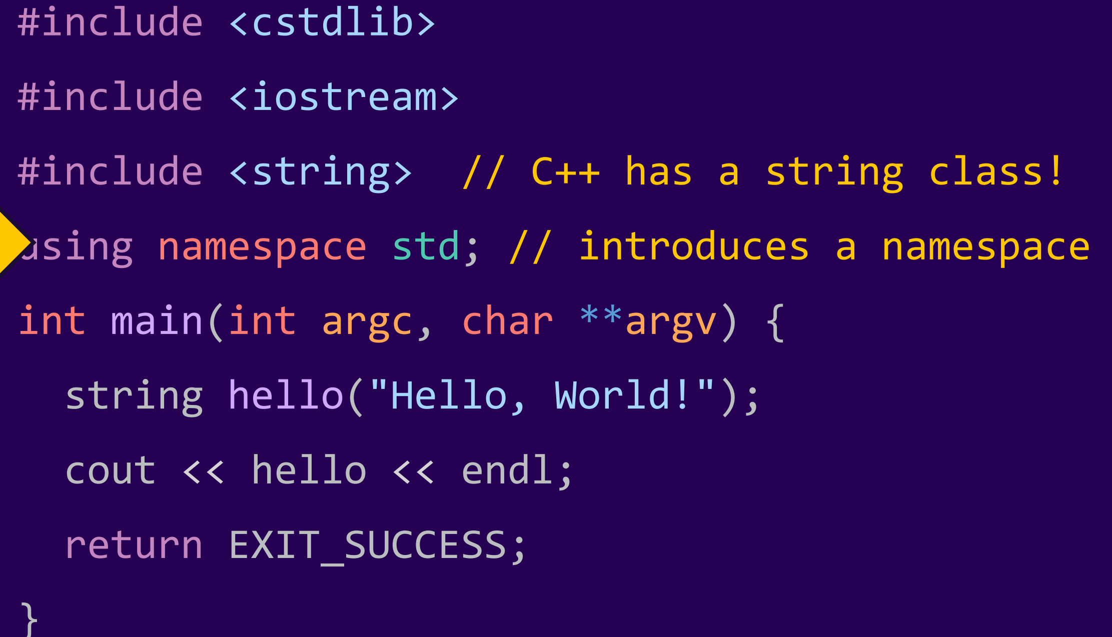
- You can still use **printf** – but **bad style** in ordinary C++ code
  - E.g Use `std::cerr` instead of `frpintf(stderr, …)`
- Can mix C and C++ idioms if needed to work with existing code, but avoid mixing if you can
  - **Use C++(17)**

```
#include <cstdio>     // for printf
#include <cstdlib>    // for EXIT_SUCCESS

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

Fancier HelloWorld.cc - namespaces

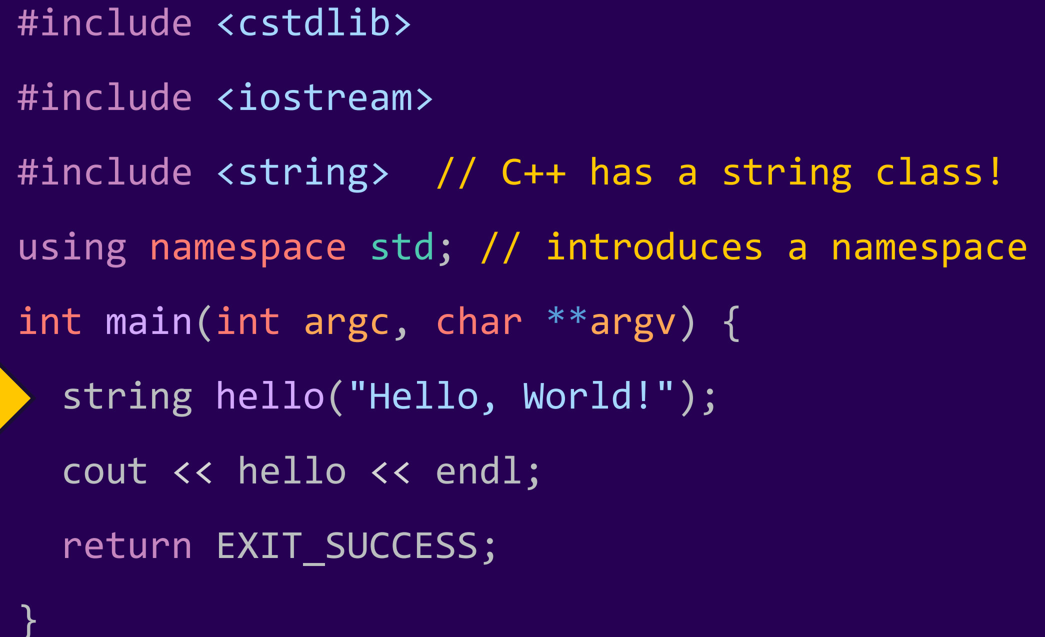
- The **`using`** keyword introduces a namespace (or part of) into the current region
- **`using namespace std;`** imports all names from **`std::`**
  - Linter will complain, but we will ignore for this class
  - Can use std::string as string, std::cout as cout, std::endl as endl
- **`Using std::cout;`** imports *only* **`std::cout`** (used as **`cout`**)



Fancier HelloWorld.cc - strings

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

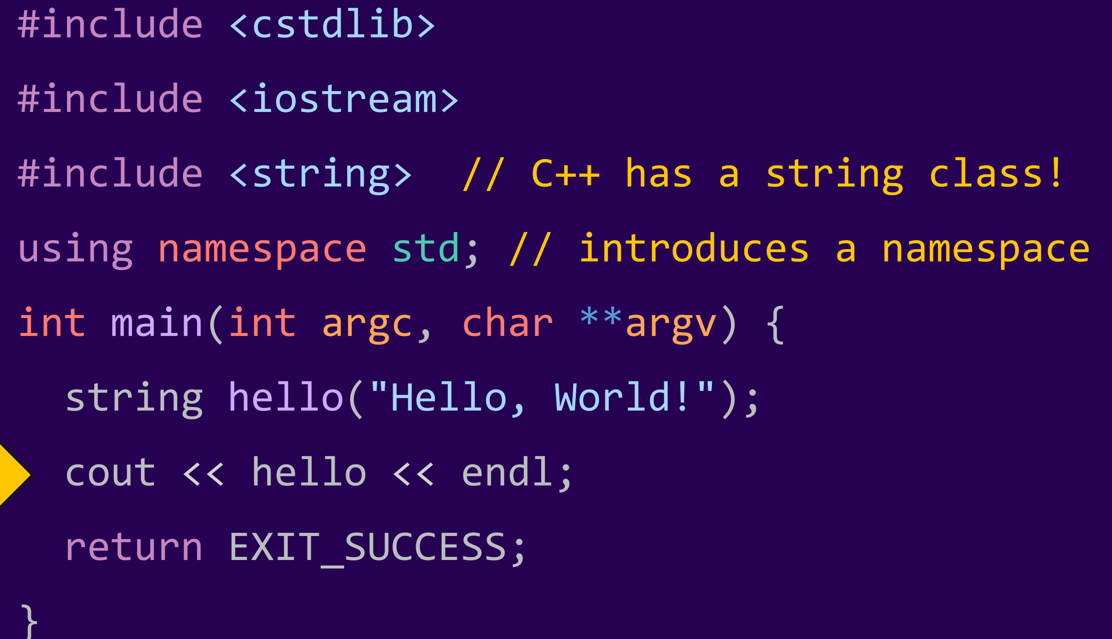
- Include the string header to use it
  - http://www.cplusplus.com/reference/string/
- Here we are instantiating a std::string object *on the **stack*** (an ordinary local variable)
- Passing the C string "Hello, World!" to its constructor method
- hello is deallocated (and its destructor invoked) when **main** returns



```
#include <cstdlib>
#include <iostream>
#include <string>  // C++ has a string class!
using namespace std; // introduces a namespace
int main(int argc, char **argv) {
  string hello("Hello, World!");
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

Fancier HelloWorld.cc – string output

- The C++ string library also overloads the << operator
- Defines a function (*not* an object method) that is invoked when the LHS is ostream and the RHS is std::string
  - http://www.cplusplus.com/reference/string/string/operator<</



```
#include <cstdlib>
#include <iostream>
#include <string>  // C++ has a string class!
using namespace std; // introduces a namespace
int main(int argc, char **argv) {
  string hello("Hello, World!");
  cout << hello << endl;
  return EXIT_SUCCESS;
}
```

Fancier HelloWorld – string operations

- The string class overloads the "=" operator
  - Copies the RHS and replaces the string's contents with it
- The string class overloads the "+" operator
  - Creates and returns a new string that is the concatenation of the LHS and RHS
- This statement is complex!
  - First "+" creates a string that is the concatenation of hello's current contents and ", World!"
  - Then "=" creates a copy of the concatenation to store in hello
  - Without the syntactic sugar:

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

```
#include <cstdlib>
#include <iostream>
using std::cout;
using std::endl;
using std::string;

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

Some more cool things about C++

- C++ primitive types include the familiar ones from C:
  - `char`, `short`, `int`, `long`, `float`, `double`, etc.
  - Also include `bool`
- C++ still has pointers, and they work the same way as in C
  - But, also, you can use **references**

Pointer Review

A **pointer** is a variable containing an address

- Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
- These work the same in C and C++

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

Pointer Review 2

x	5
y	10
z	

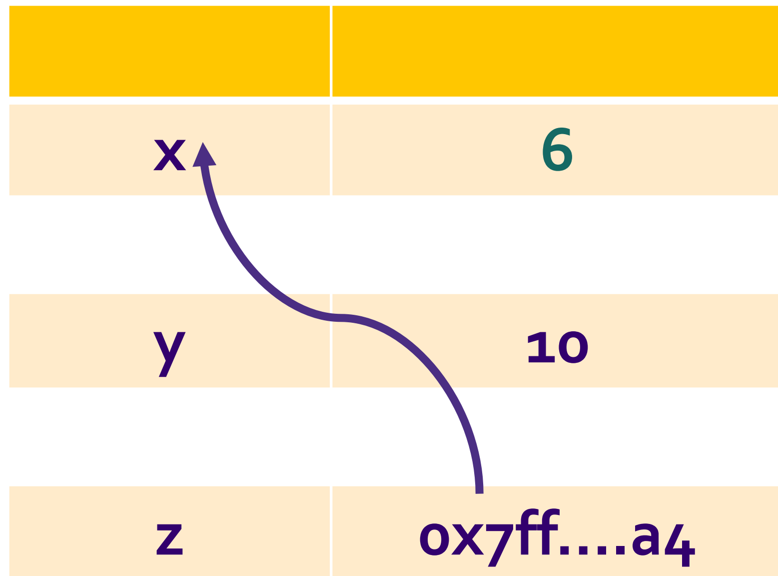
```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

Pointer Review 3

x	5
y	10
z	0x7ff….a4

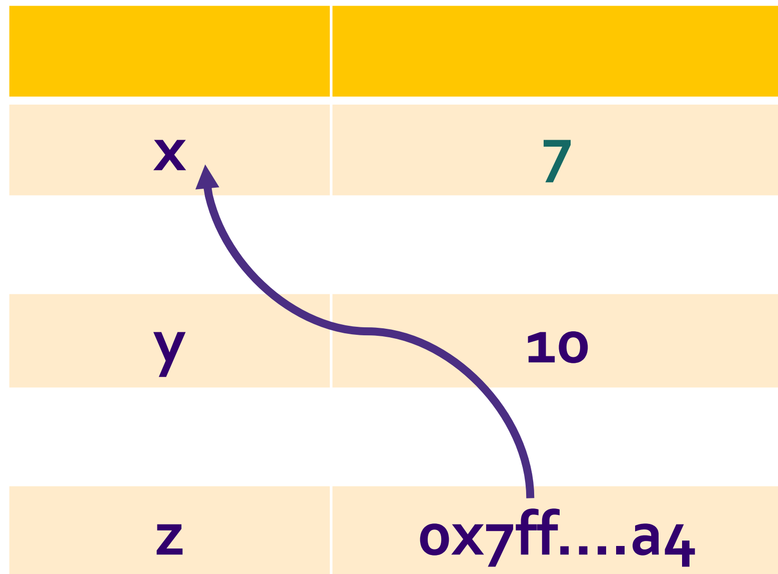
```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

Pointer Review 4



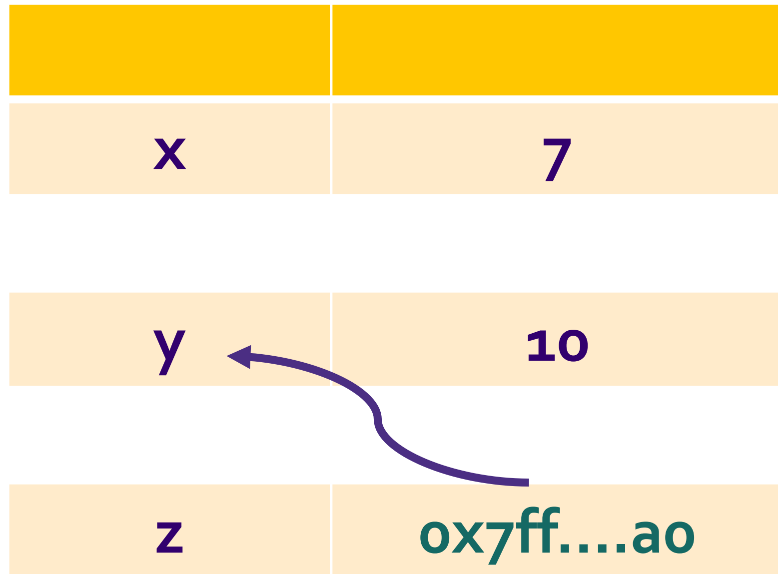
```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

Pointer Review 5



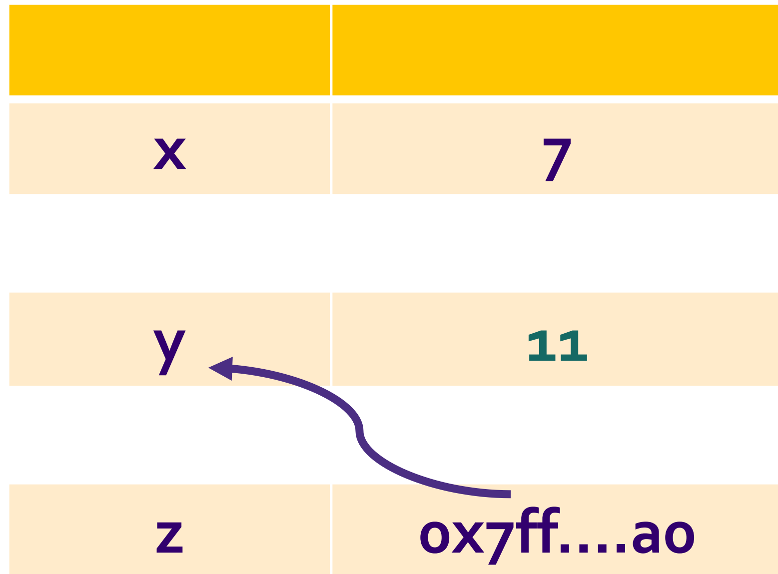

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

Pointer Review 6




```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

Pointer Review 7




```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

References

A **reference** is an alias for another variable

- *Alias*: another name that is bound to the aliased variable - No new storage!
  - Mutating a reference ***is*** mutating the aliased variable
- Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;
  z += 1;
  x += 1;
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

References 2

x	5
y	10
z	

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int &z = x;
  z += 1;
  x += 1;
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

References 3

x, z	5
y	10

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int &z = x;
  z += 1;
  x += 1;
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

References 4

x, z	6
y	10

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int &z = x;
  z += 1;
  x += 1;
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

References 5

x, z	7
y	10

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int &z = x;
  z += 1;
  x += 1;
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

References 6

x, z	10
y	10

This is different: z is bound to x, so this is just a normal assignment operator.
You can't 'rebind' a reference, so it must be initialized when it is declared.

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int &z = x;
  z += 1;
  x += 1;
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

References 7

x, z	11
y	10

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int &z = x;
  z += 1;
  x += 1;
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

Pointer syntax

`&` and `*` are used as both an operator in an expression and as part of a declaration. The context in which a symbol is used determines what the symbol means:

- `int i = 42;`
- `int& r = i;` // & follows a type and is part of a declaration; r is a reference
- `int* p;` // * follows a type and is part of a declaration; p is a pointer
- `p = &i;` // & is used in an expression as the address-of operator
- `*p = i;` // * is used in an expression as the dereference operator
- `int& r2 = *p;` // & is part of the declaration; * is the dereference operator

In declarations, & and * are used to form *compound* types. In expressions, these same symbols are used to denote an operator.

Pass by reference

A **reference** is an alias for another variable

- *Alias*: another name that is bound to the aliased variable - No new storage!
  - Mutating a reference ***is*** mutating the aliased variable
- Introduced in C++ as part of the language

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
int main(int argc, char** argv) {
  int a = 5, b = 10;
  swap(a, b);
  cout<< "a "<< a << "b "<< b <<endl;
  return EXIT_SUCCESS;
}
```

Pass by reference 2

a	5
b	10

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
int main(int argc, char** argv) {
  int a = 5, b = 10;
  swap(a, b);
  cout<< "a "<< a << "b "<< b <<endl;
  return EXIT_SUCCESS;
}
```

Pass by reference 3

a, x	5
b, y	10

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
int main(int argc, char** argv) {
  int a = 5, b = 10;
  swap(a, b);
  cout<< "a "<< a << "b "<< b <<endl;
  return EXIT_SUCCESS;
}
```

Pass by reference 4

a, x	5
b, y	10
tmp	5

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
int main(int argc, char** argv) {
  int a = 5, b = 10;
  swap(a, b);
  cout<< "a "<< a << "b "<< b <<endl;
  return EXIT_SUCCESS;
}
```

Pass by reference 5

a, x	10
b, y	10
tmp	5

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
int main(int argc, char** argv) {
  int a = 5, b = 10;
  swap(a, b);
  cout<< "a "<< a << "b "<< b <<endl;
  return EXIT_SUCCESS;
}
```

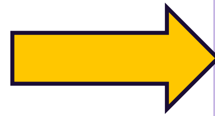
Pass by reference 6

a, x	10
b, y	5
tmp	5

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
int main(int argc, char** argv) {
  int a = 5, b = 10;
  swap(a, b);
  cout<< "a "<< a << "b "<< b <<endl;
  return EXIT_SUCCESS;
}
```

Pass by reference 7

a	10
b	5



```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
int main(int argc, char** argv) {
  int a = 5, b = 10;
  swap(a, b);
  cout<< "a "<< a << "b "<< b <<endl;
  return EXIT_SUCCESS;
}
```

Using Pointers or References

C

- Pointers can be used to access variables outside a function
  - Save memory by not copying large data structures
  - Change values without having to return more than one variable

C++

- User references for this purpose