CSE 374 Programming Concepts and Tools

Lecture 17 - Intro to C++

Exam 2

Next Time

C++ Classes

Today

Today: Intro to C++

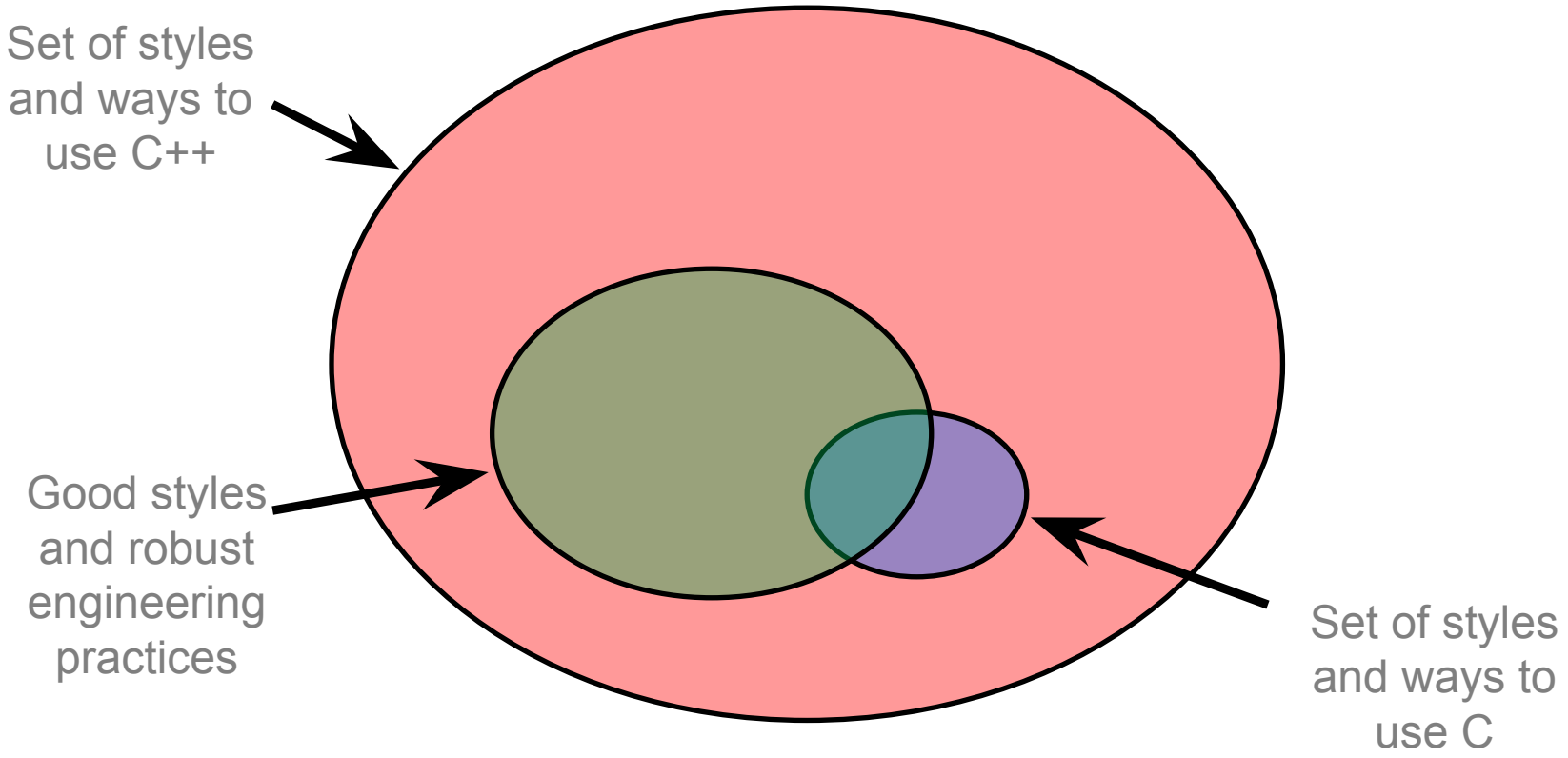
- What is C++? Why learn it?
- Differences from C
- Hello World in C++
- Pointers vs. References (Revenge of the `&`)
- HW6: T9 Implementation Overview

C++ Intro

Why C++?

- Originally created as an extension to C to add object-oriented programming
  - Bjarne Stroustrup (creator of C++) wanted a language both fast (like C) but with other useful features
- OOP can be a powerful tool for *abstraction*
  - And C doesn't have OOP
  - Abstractions make it easier to organize code
- Still pretty old: first released in 1985
- Provides more programming assistance
  - C's standard library is very lightweight
  - C++'s standard template library (STL) provides *generic* implementations of:
    - popular data structures
    - algorithms
    - iterators
- Like C: C++ doesn't do a lot of hand-holding

How to Think About 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;
}
```

- Compile with gcc:

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

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

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

`*.cpp` is also a common file extension for C++ files, but we'll use `.cc` in this class

Hello World in C vs. 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;
}
```

helloworld.**c**

```cpp
#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;
}
```

helloworld.**cc**

Example: Hello World in 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;
}
```

`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`

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

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 instance declared by `iostream`, living within the "`std`" namespace

- C++'s name for `stdout`, `std::cout` is an object of class `ostream`
  - **o**utput **stream**
- 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>
#include <cstdlib>

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

C++ has a stronger distinction 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 🎉🎉🎉
  - Use it!

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

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;
}
```

ostream object

still a `char*`

`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>
#include <cstdlib>

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

This is equivalent to:

```
std::cout <<
"Hello,world!";
std:cout <<
std::endl;
```

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>
#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
- If you need to print a `'\n'`, you should probably use `std::endl`

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 complex
  - It's powerful, but really dangerous
  - Once you mix everything together (templates, operator overloading, method overloading, multiple inheritance), it can get *really* hard to know what's actually happening!

Let's Refine It a Bit

helloworld2.cc

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

Let's Refine It a Bit

helloworld2.cc

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS
#include <string>      // for 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::`
  - Linter will complain, but we will ignore for this class
- `using std::cout;` imports *only* `std::cout` (used as `cout`)

Let's Refine It a Bit

helloworld2.cc

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

using namespace std;

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

Benefits of `using namespace std;`

- We can now refer to `std::string` as `string`, `std::cout` as `cout`, and `std::endl` as `endl`

Let's Refine It a Bit

helloworld2.cc

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

using namespace std;

int main(int argc, char** argv) {
  string hello("Hello, World!");
  cout << hello << 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 method
- `hello` is deallocated (and its destructor invoked) when **`main`** returns

Let's Refine It a Bit

helloworld2.cc

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

using namespace std;

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

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<</

String Concatenation

concat.cc

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

using namespace std;

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

The string class overloads the “+” operator

- Creates and returns a new string that is the concatenation of the LHS and RHS

String Assignment

concat.cc

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

using namespace std;

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

The string class overloads the “=” operator

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

String Manipulation

concat.cc

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

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!"));`

Operators are just member functions

C and C++

helloworld3.cc

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

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
  - E.g Use `std::cerr` instead of `fprintf(stderr, …)`
- Can mix C and C++ idioms if needed to work with existing code, but avoid mixing if you can
  - **Use C++(17)**

Reading

echonum.cc

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

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”
  - Can be used in conditionals ! (`std::cin >> num`) is `true` if successful
- Has a `getline()` method and methods to detect and clear errors

Demo: `echonum.cc`

C++ References

Review: Pointer

Note: Arrow points to *next* instruction.

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;
}
```

x	5
y	10
z	

pointer.cc

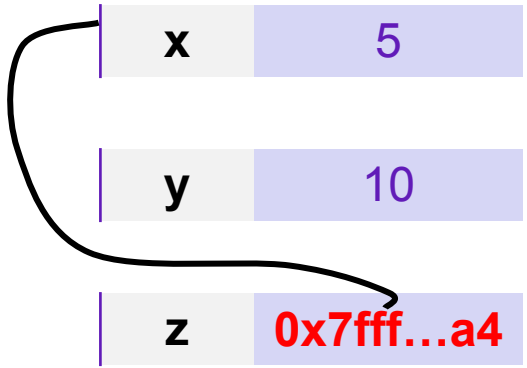
Review: Pointer

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.cc

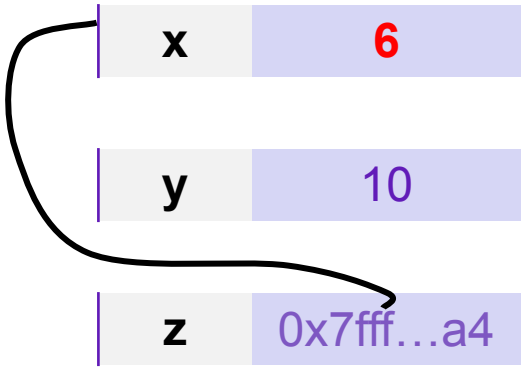
Review: Pointer

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;  // sets x to 6
  x += 1;
  z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



pointer.cc

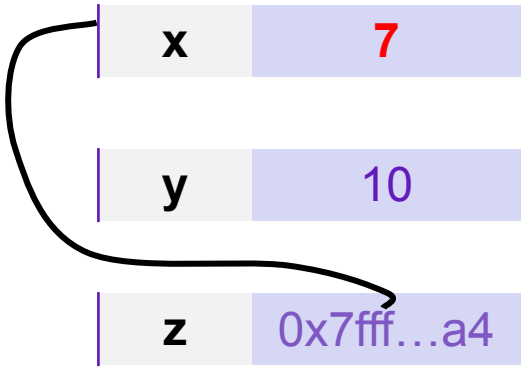
Review: Pointer

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;  // sets x to 6
  x += 1;  // sets x (and *z) to 7
  z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



pointer.cc

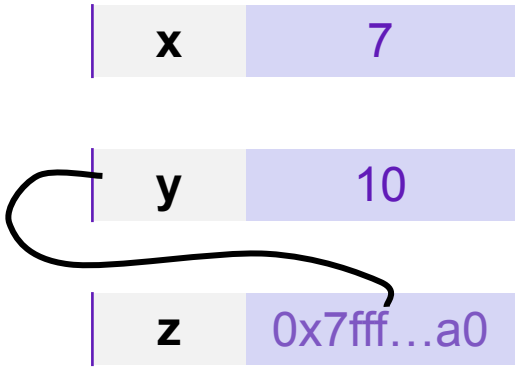
Review: Pointer

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;  // sets x to 6
  x += 1;  // sets x (and *z) to 7
  z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



pointer.cc

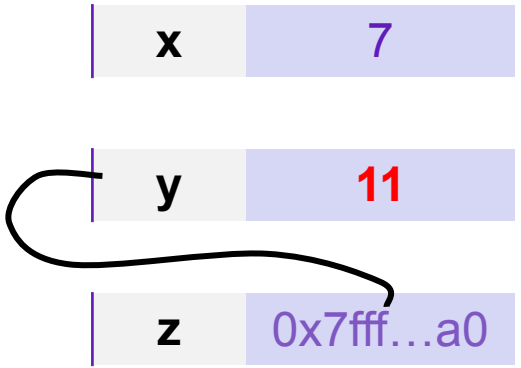
Review: Pointer

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;  // sets x to 5 + 1 = 6
  x += 1;  // sets x (and *z) to 6 + 1 = 7
  z = &y;
  *z += 1; // sets y to 10 + 1 = 11

  return EXIT_SUCCESS;
}
```



pointer.cc

References

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

- *Alias*: another name that is bound to the aliased variable
  - 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;
}
```

When we use '`&`' in a type declaration, it is a reference.

`&var` is still "address of var"

x	5

y	10

reference.cc

References

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

- *Alias*: another name that is bound to the aliased variable
  - 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; // binds the name "z" to x
  z += 1;
  x += 1;
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

x, z	5

y	10

reference.cc

References

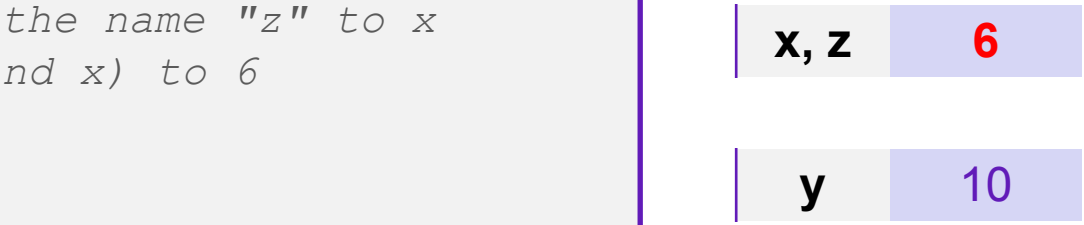
A **reference** is an alias for another variable

- *Alias*: another name that is bound to the aliased variable
  - 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; // binds the name "z" to x
  z += 1;  // sets z (and x) to 6
  x += 1;
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

reference.cc



References

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

- *Alias*: another name that is bound to the aliased variable
  - 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; // binds the name "z" to x
  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

reference.cc

References

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

- *Alias*: another name that is bound to the aliased variable
  - 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; // binds the name "z" to x
  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7
  z = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

Normal assignment

There is no way to rebind a reference to refer to a different object. Because there is no way to rebind a reference, references must be initialized.

x, z	7

y	10

reference.cc

References

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

- *Alias*: another name that is bound to the aliased variable
  - 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; // binds the name "z" to x
  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7
  z = y;  // sets z (and x) to the value of y
  z += 1;

  return EXIT_SUCCESS;
}
```

x, z	10

y	10

reference.cc

References

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

- *Alias*: another name that is bound to the aliased variable
  - 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; // binds the name "z" to x
  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7
  z = y;  // sets z (and x) to the value of y
  z += 1;  // sets z (and x) to 11

  return EXIT_SUCCESS;
}
```

reference.cc

x, z	11

y	10

Some Symbols Have Multiple Meanings

`&` 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

Pass-By-Reference

C++ allows you to use real pass-by-*reference*

- Client passes in an argument with normal syntax
  - Function uses reference parameters with normal syntax
  - Modifying a reference parameter modifies the caller's argument!

```
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;
}
```

(main) **a**	5

(main) **b**	10

passbyreference.cc

Pass-By-Reference

C++ allows you to use real pass-by-*reference*

- Client passes in an argument with normal syntax
  - Function uses reference parameters with normal syntax
  - Modifying a reference parameter modifies the caller's argument!

```
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;
}
```

Parameters are attached to variables provided by caller

(main) **a**	5

(main) **b**	10

passbyreference.cc

Pass-By-Reference

C++ allows you to use real pass-by-*reference*

- Client passes in an argument with normal syntax
  - Function uses reference parameters with normal syntax
  - Modifying a reference parameter modifies the caller's argument!

```
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;
}
```

passbyreference.cc

(main) **a** (swap) **x**	5
(main) **b** (swap) **y**	10
(swap) **tmp**	

Pass-By-Reference

C++ allows you to use real pass-by-*reference*

- Client passes in an argument with normal syntax
  - Function uses reference parameters with normal syntax
  - Modifying a reference parameter modifies the caller's argument!

```
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;
}
```

passbyreference.cc

(main) **a** (swap) **x**	5
(main) **b** (swap) **y**	10
(swap) **tmp**	**5**

Pass-By-Reference

C++ allows you to use real pass-by-*reference*

- Client passes in an argument with normal syntax
  - Function uses reference parameters with normal syntax
  - Modifying a reference parameter modifies the caller's argument!

```
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;
}
```

passbyreference.cc

(main) **a** (swap) **x**	**10**
(main) **b** (swap) **y**	10
(swap) **tmp**	5

Pass-By-Reference

C++ allows you to use real pass-by-*reference*

- Client passes in an argument with normal syntax
  - Function uses reference parameters with normal syntax
  - Modifying a reference parameter modifies the caller's argument!

```
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;
}
```

passbyreference.cc

(main) **a** (swap) **x**	10
(main) **b** (swap) **y**	**5**
(swap) **tmp**	5

Pass-By-Reference

C++ allows you to use real pass-by-*reference*

- Client passes in an argument with normal syntax
  - Function uses reference parameters with normal syntax
  - Modifying a reference parameter modifies the caller's argument!

```
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;
}
```

(main) **a**	10

(main) **b**	5

passbyreference.cc

Best Practices

C Programming

- Uses *pointers* as function arguments to access and modify objects outside the function

C++ Programming

- Prefers *references* as function arguments instead
  - More intuitive syntax and reduced risk of pointer-related errors

Polling Time!

Please answer in the
Poll: Lec 17 on Gradescope

What will this snippet of code output?

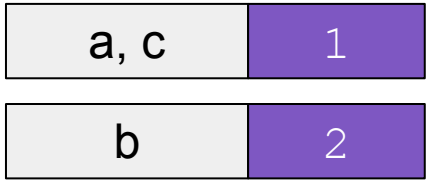
First step: Figure out what values `a`, `b`, `c` will have after calling `foo`

Then put it all together:
`(a, b, c)`

```
void foo(int& x, int* y, int z) {
    z = *y;
    x += 2;
    y = &x;
}
int main(int argc, char** argv) {
    int a = 1;
    int b = 2;
    int& c = a;

    foo(a, &b, c);
    std::cout << "(" << a << ", " << b
    << ", " << c << ")" << std::endl;
    return EXIT_SUCCESS;
}
```

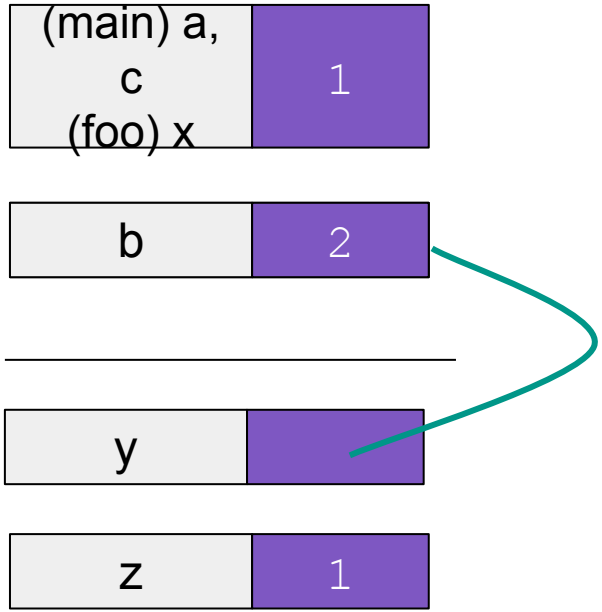
Poll Question Explained




```
void foo(int& x, int* y, int z) {
    z = *y;
    x += 2;
    y = &x;
}
int main(int argc, char** argv) {
    int a = 1;
    int b = 2;
    int& c = a;

    foo(a, &b, c);
    std::cout << "(" << a << ", " << b
    << ", " << c << ")" << std::endl;
    return EXIT_SUCCESS;
}
```

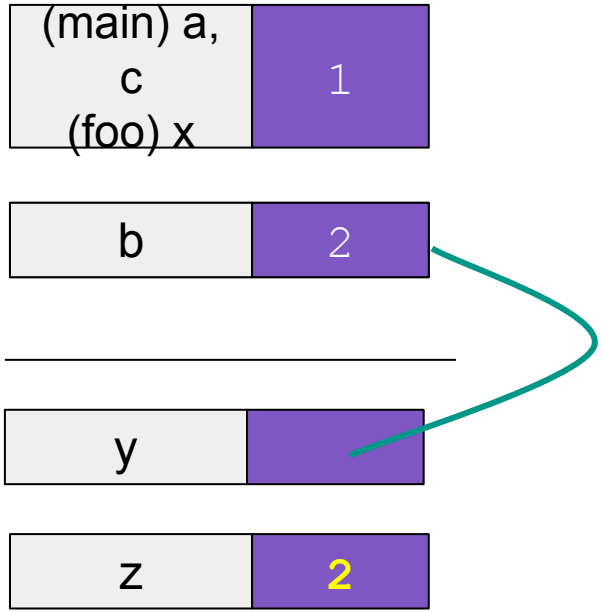
Poll Question Explained




```
void foo(int& x, int* y, int z) {
    z = *y;
    x += 2;
    y = &x;
}
int main(int argc, char** argv) {
    int a = 1;
    int b = 2;
    int& c = a;

    foo(a, &b, c);
    std::cout << "(" << a << ", " << b
    << ", " << c << ")" << std::endl;
    return EXIT_SUCCESS;
}
```

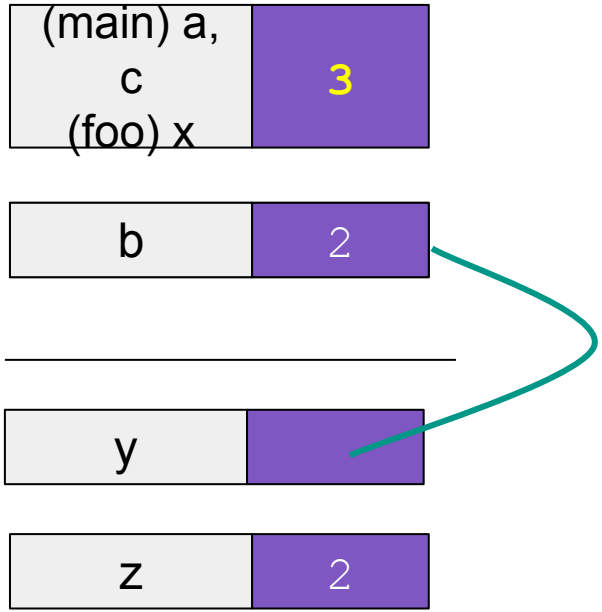
Poll Question Explained




```
void foo(int& x, int* y, int z) {
    z = *y;
    x += 2;
    y = &x;
}
int main(int argc, char** argv) {
    int a = 1;
    int b = 2;
    int& c = a;

    foo(a, &b, c);
    std::cout << "(" << a << ", " << b
    << ", " << c << ")" << std::endl;
    return EXIT_SUCCESS;
}
```

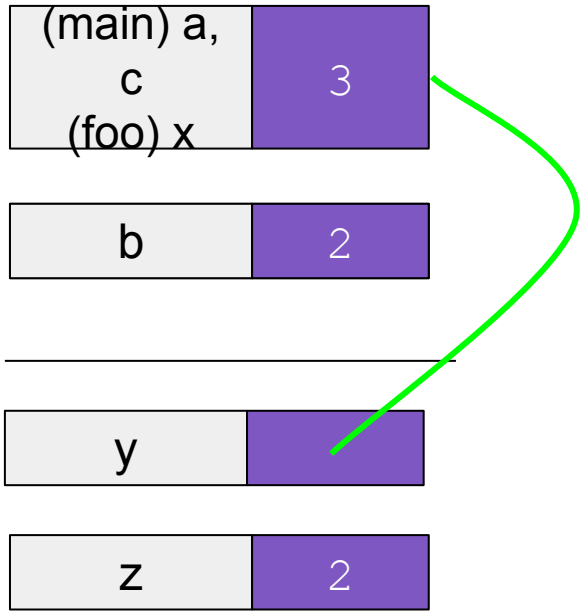
Poll Question Explained




```
void foo(int& x, int* y, int z) {
    z = *y;
    x += 2;
    y = &x;
}
int main(int argc, char** argv) {
    int a = 1;
    int b = 2;
    int& c = a;

    foo(a, &b, c);
    std::cout << "(" << a << ", " << b
    << ", " << c << ")" << std::endl;
    return EXIT_SUCCESS;
}
```

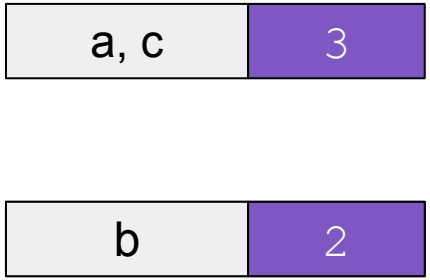
Poll Question Explained




```
void foo(int& x, int* y, int z) {
    z = *y;
    x += 2;
    y = &x;
}
int main(int argc, char** argv) {
    int a = 1;
    int b = 2;
    int& c = a;

    foo(a, &b, c);
    std::cout << "(" << a << ", " << b
    << ", " << c << ")" << std::endl;
    return EXIT_SUCCESS;
}
```

Poll Question Explained



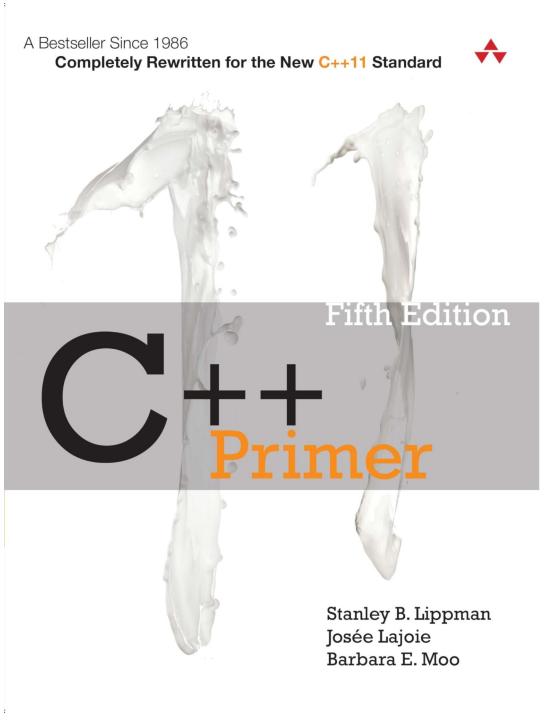

```
void foo(int& x, int* y, int z) {
    z = *y;
    x += 2;
    y = &x;
}
int main(int argc, char** argv) {
    int a = 1;
    int b = 2;
    int& c = a;

    foo(a, &b, c);
    std::cout << "(" << a << ", " << b
    << ", " << c << ")" << std::endl;
    return EXIT_SUCCESS;
}
```

Aside: C++ Primer

It's hard to learn the "why is it done this way" from reference docs, and even harder to learn from random stuff on the web

- Lectures and examples will introduce the main ideas, but aren't everything you'll need to understand
- **Free** access through UW libraries



HW5: Implementation

Assignment Reminders

EX15 due before next class

Exam 2 is on Friday

- You ***must*** sign up for either the morning or afternoon session!
- https://docs.google.com/spreadsheets/d/1p6pn-MqDTJMltZaYD6ilOni2FlxUgiRTv-wc_fU7Nck/edit?gid=0#gid=0

Tons more on **C++** programming next week! 👀🥳