CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

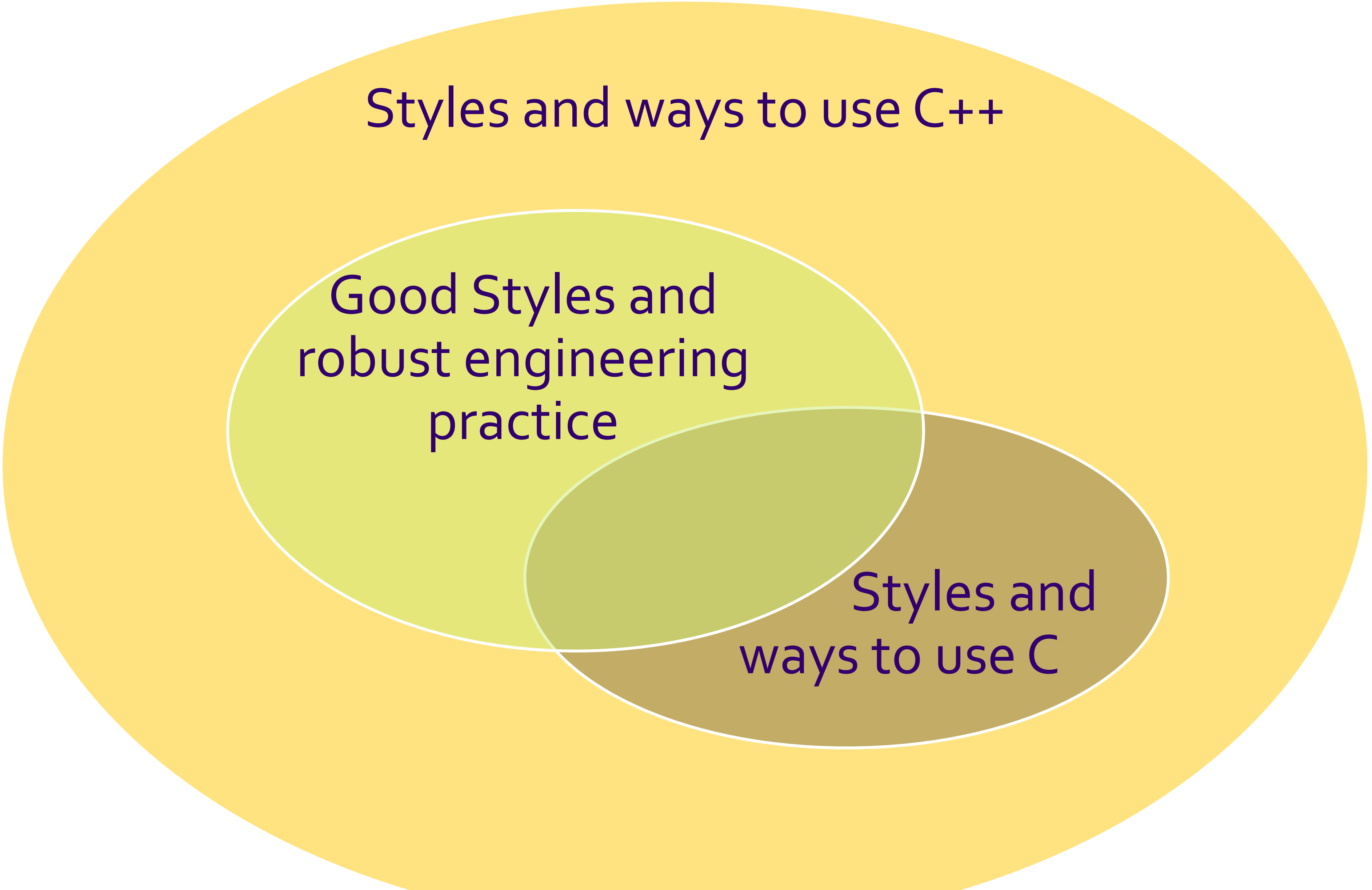
Today's Goals

Subject Matter

- Introduction to C++
  - References
  - Const
  - New & Delete
  - Classes

Your Goals

- Work through reference example
- HW6


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

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`

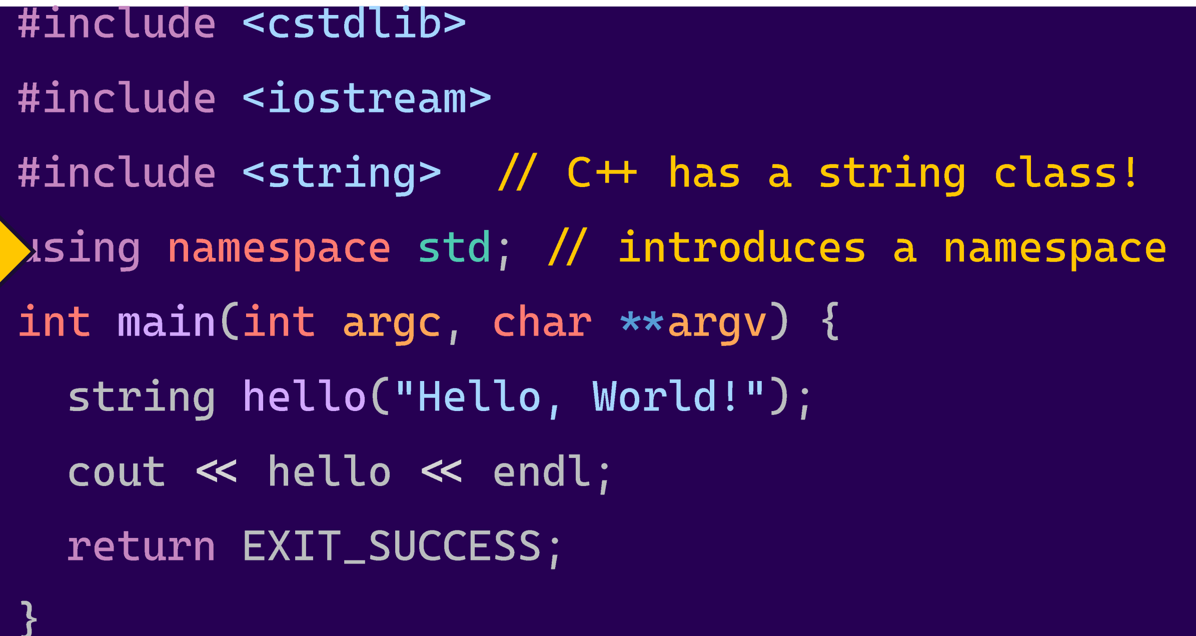
```
#include <iostream> // for cout, endl
#include <cstdlib> // for EXIT_SUCCESS

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

Hello World in C++: 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

Fancier HelloWorld.cc - namespaces

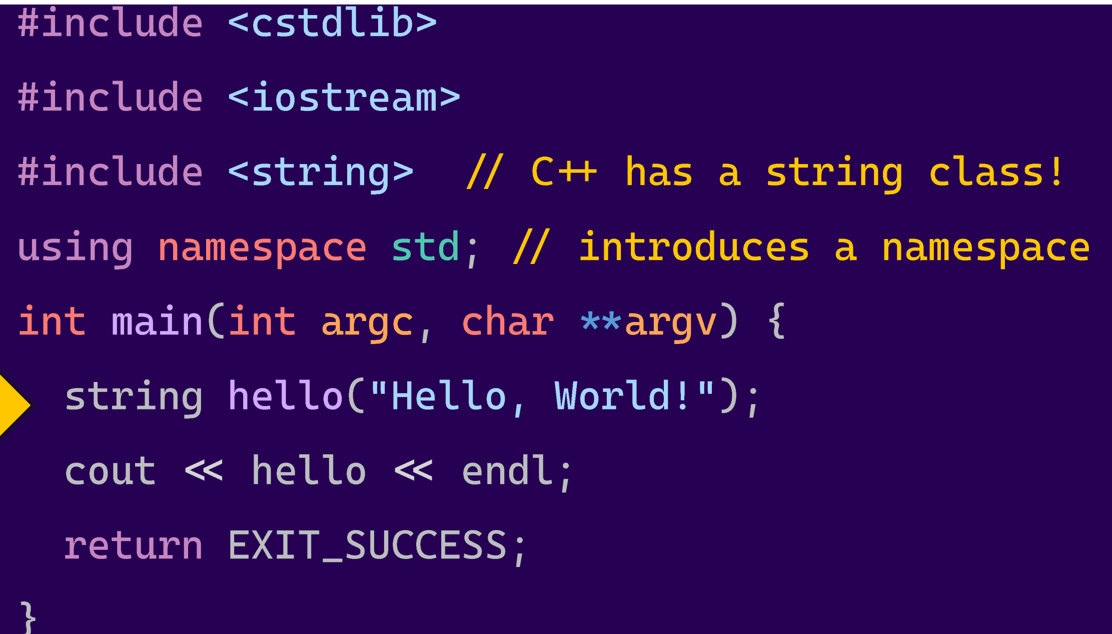
- The `using` keyword introduces a namespace (or part of) into the current region
- `using namespace std;` imports all names from `std::`
  - Linter will complain, but we will ignore for this class
  - Can use std::string as string, std::cout as cout, std::endl as endl
- `Using std::cout;` imports *only*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



Some more cool things about C++

- **C++ still uses the C pre-processor**
  - That means that pre-processing (includes, defines, etc) works the same way!
- **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	

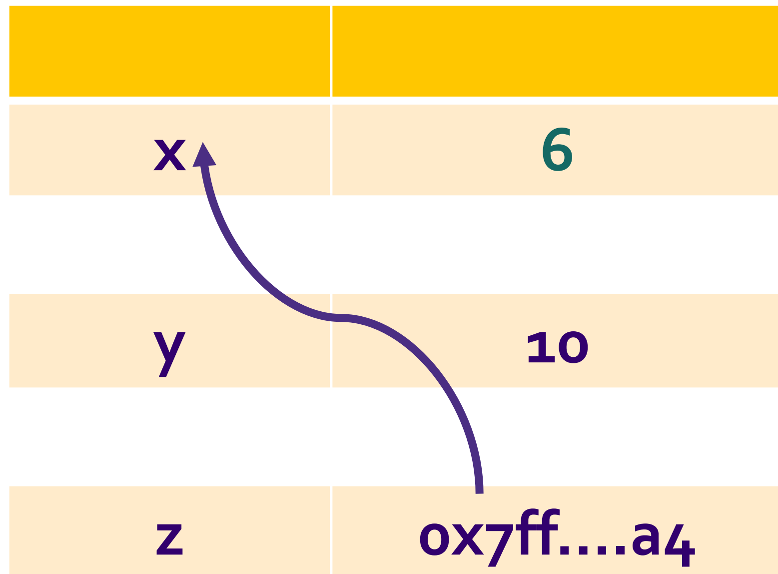
```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

Pointer Review 3

x	5
y	10
z	0x7ff....a4

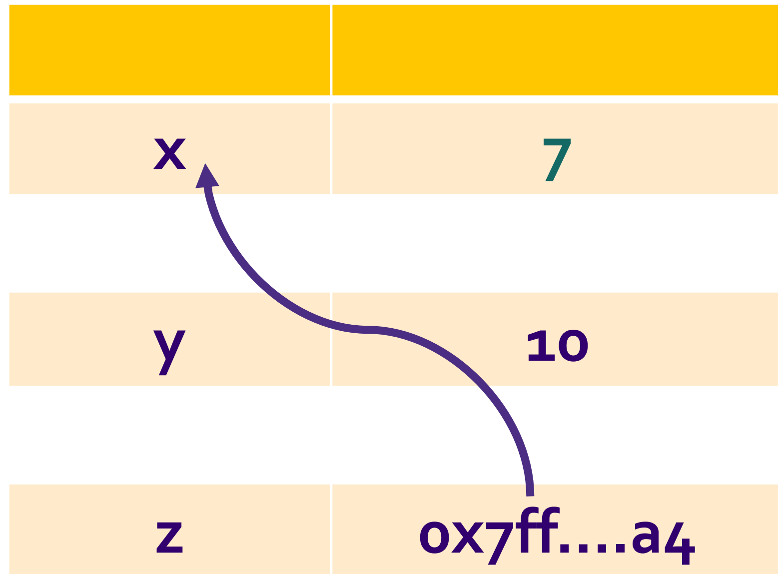
```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

Pointer Review 4



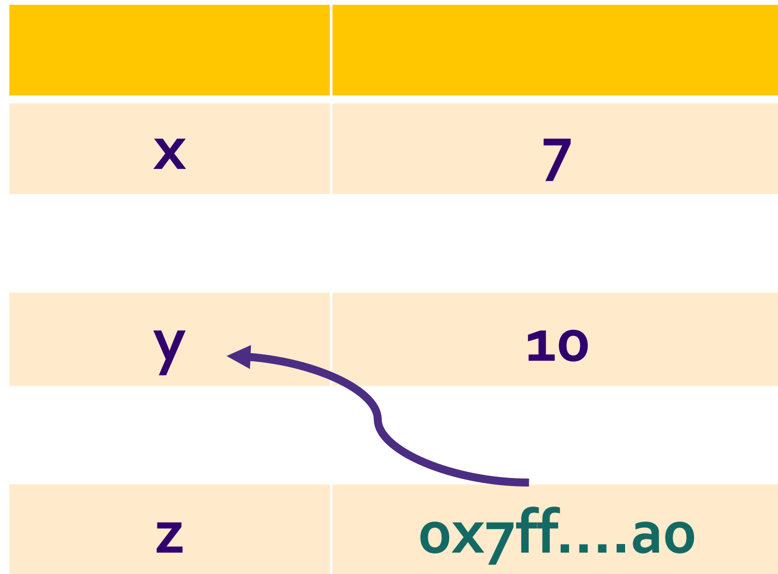
```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

Pointer Review 5



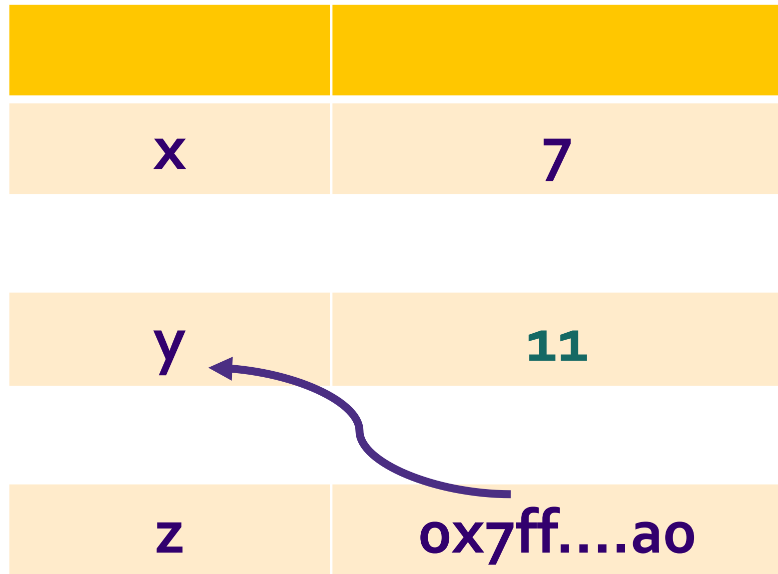
```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;
  *z += 1;
   x += 1;
   z = &y;
  *z += 1;
  return EXIT_SUCCESS;
}
```

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 `const`

The *qualifier* `const` labels a variable as '**unchangeable**' or '**unmutable**'

- Exists and C and C++
  - Used much more in C++
- Produces compile time errors
- Used for defensive programming

```
void BrokenPrintSquare(const int& i) {
  i = i*i;  // compiler error here!
  std::cout << i << std::endl;
}


int main(int argc, char** argv) {
  int j = 2;
  BrokenPrintSquare(j);
  return EXIT_SUCCESS;
}
```

Pointers & `const`

- Pointers can change data in two different contexts:
  - You can change the value of the pointer
  - You can change the thing the pointer points to (via dereference)
- const can be used to prevent either/both of these behaviors!
  - const next to pointer name means you can't change the value of the pointer
    - **int* const ptr;** // cannot change the value of ptr
  - const next to data type pointed to means you can't use this pointer to change the thing being pointed to
    - **const int* ptr** // cannot change the value of *ptr
- Tip: read variable declaration from *right-to-left*

```
int main(int argc, char** argv) {
  int x = 5;              // int
  const int y = 6;        // (const int)
  y++;

  // pointer to a (const int)
  const int* z = &x;
  *z += 1;
  z++;

  // (const pointer) to a (variable int)
  int* const w = &x;
  *w += 1;
  w++;

  // (const pointer) to a (const int)
  const int* const v = &x;
  *v += 1;
  v++;

  return EXIT_SUCCESS;
}
```

Pointers & `const` compiled

- Pointers can change data in two different contexts:
  - You can change the value of the pointer
  - You can change the thing the pointer points to (via dereference)
- const can be used to prevent either/both of these behaviors!
  - const next to pointer name means you can't change the value of the pointer
    - **int* const ptr;** // cannot change the value of ptr
  - const next to data type pointed to means you can't use this pointer to change the thing being pointed to
    - **const int* ptr** // cannot change the value of *ptr
- <u>Tip</u>: read variable declaration from *right-to-left*

```
int main(int argc, char** argv) {
  int x = 5;                // int

  const int y = 6;          // (const int)
  y++;                      // Compiler ERROR

  // pointer to a (const int)
  const int* z = &x;
  *z += 1;                  // Compiler ERROR
  z++;                      // OK

  // (const pointer) to a (variable int)
  int* const w = &x;
  *w += 1;                  // OK
  w++;                      // Compiler ERROR

  // (const pointer) to a (const int)
  const int* const v = &x;
  *v += 1;                  // Compiler ERROR
  v++;                      // Compiler ERROR

  return EXIT_SUCCESS;
}
```

Parameters & `const`

- A const parameter *cannot* be mutated inside the function
  - Therefore, it does not matter if the argument can be mutated or not

- A non-const parameter *may* be mutated inside the function
  - It would be BAD if you passed it a const variable
  - Compiler won't let you pass in const parameters

```
void foo(const int* y) {
  std::cout << *y << std::endl;
}

void bar(int* y) {
  std::cout << *y << std::endl;
}

int main(int argc, char** argv) {

  const int a = 10;
  int b = 20;

  foo(&a);    // OK
  foo(&b);    // OK
  bar(&a);    // not OK - error
  bar(&b);    // OK

  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++

Use references for this purpose. Google C++ style guide suggests

- Input parameters:
  - Either use values (for primitive types like int or small structs/objects)
  - Or use const references (for complex struct/object instances)
- Output parameters:
  - Use const pointers: unchangeable pointers referencing changeable data
- Ordering:
  - List input parameters first, then output parameters last

```
void CalcArea(const int& width, const int& height, int* const area) {
  *area = width * height;
}
```

WHAT'S NEW?

Memory allocation and deallocation in C++

New / Delete

C

```
int* x = (int*) malloc(sizeof(int));
int* arr = (int*) malloc(sizeof(int) * 100);
free(x);
free(arr);
```

C++ : Nicer syntax for a similar purpose

```
int* x = new int(4);        // x stores the value 4.
int* arr = new int[100];
delete x;
delete [] arr;
```

New

To allocate on the heap using C++, you use the `new` keyword instead of `malloc()`

- Use `new` to allocate an object (*e.g.* `new Point`), or a primitive type (*e.g.* `new int`)
- When allocating you can specify a constructor or initial value
  - (e.g. `new Point(1, 2)`) or (e.g. `new int(333)`)
- `new` is an operator – it allocates memory and then calls a constructor if appropriate.
  - If you do not specify initialization uses default constructor
  - Throws an exception when it fails (does not return NULL)
  - Returns a pointer of the desired type
  - Size calculated by compiler, not user

```
#include "Point.h”
using namespace std;
int main() {
  Point* x = new Point(x,y);
  int* y = new int;
  y = 3;
  cout<< “x.x: " << x->get_x() << endl;
  cout<< "y: "<<y<< “,*y: "<< *y<<endl;
  delete x;
  delete y;
  return EXIT_SUCCESS;
}
```

Delete

- To deallocate a heap-allocated object or primitive, use the `delete` keyword instead of `free`
- `delete` calls a destructor if necessary
- Don't mix and match!
  - *Never* `free()` something allocated with `new`
  - *Never* `delete` something allocated with `malloc()`
  - Careful if you're using a legacy C code library or module in C++
  - If you delete already deleted memory, then you will get undefined behavior. (Same as when you double free in c)

```
#include "Point.h"
using namespace std;
int main() {
  Point* x = new Point(x,y);
  int* y = new int;
  y = 3;
  cout<< "x.x: " << x->get_x() << endl;
  cout<< "y: "<<y<< ",*y: "<< *y<<endl;
  delete x;
  delete y;
  return EXIT_SUCCESS;
}
```

Malloc v. New

Malloc ()	**new**
A function	An operator / key word
Used in C often, in C++ rarely	Used in C never, in C++ often
Allocates memory for anything	Allocates memory arrays, objects, primitives
Returns a `void*` (should be cast)	Returns a `T*` (does not need a cast)
Returns `NULL` when out of memory	Throws an exception when out of memory
Deallocate with `free()`	Deallocate with `delete` or `delete[]`

Null v. nullptr

- C and C++ have long used `NULL` as a pointer value that references nothing
- C++11 introduced a new literal for this: **`nullptr`**
- New reserved word
- Interchangeable with `NULL` for all practical purposes, but it has type `T*` for any/every `T`, and is not an integer value
  - Still can convert to/from integer 0 for tests, assignment, etc.
- Advice: prefer **`nullptr`** in C++11 code
  - Though NULL will also be around for a long, long time

C++ Array allocation

To dynamically allocate an array: Default initialize: `type* name = new type[size];`

To dynamically deallocate an array: `delete[] name;`

- It is an *incorrect* to use `delete name;` on an array
  - The compiler probably won't catch this, though (!) because it can't always tell if `name*` was allocated with `new type[size];` or `new type;`
    - Especially inside a function where a pointer parameter could point to a single item or an array and there's no way to tell which!
  - Result of wrong delete is undefined behavior

```
int main() {

  int stack_arr[3]; // stack un-set
  int* heap_arr = new int[3]; // heap un-set
  int* heap_arr_init_val = new int[3]();  // heap(0, 0, 0)
  int* heap_arr_init_lst = new int[3]{4, 5};  // C++11 syntax, heap(4, 5, 0)

   ...
  delete heap_arr;              // BAD
  delete[] heap_arr_init_val;   // ok
```

CLASS ACT

C++ Classes

Structs & Classes

- C language allows structs
  - Puts related data in the same place
  - Allows access to fields by name
  - but offer no other functionality
- C++ – C with Classes!
  - Puts related data in the same place
  - Allows access to fields by name
  - Allows self-contained method definition
  - Allows for abstraction
- Actually, C++ still allows structs
  - Like classes except fields are public by default

```
typedef struct Point {
  int x;
  int y;
} Point;

Point makePoint (int x, int y)


// Class that represents a two-dimensional integral coordinate plane
class Point {
 public:
  Point(int x, int y);  // constructor
  int get_x() const { return x_; }  // inline member function
  int get_y() const { return y_; }  // inline member function
  double Distance(const Point &p) const;  // member function
  void SetLocation(int x, int y);  // member function


 private:
  int x_;  // data member
  int y_;  // data member
};  // class Point
```

Struct v. Class

C Struct	C++ Class (also C++ struct)
Only contains data fields	Contains data and method fields
Fields are always accessible	Access modifiers allow private or protected fields
Does not contain methods	Contains methods to act on member fields
Not self-contained	Self-contained
Does not promote abstraction (can not be a parent)	Used for abstraction, polymorphism, etc.

Class layouts

- Classes are often contained in namespaces.
  - Multiple classes could be in the same namespace
- Members are private by default
  - Declare public or protected as necessary
- Usually put constructors, then destructors, then other functions
- Google style guide (note, doesn't match this example)
  - Put public then protected then private
  - Fields are private and go last

```
namespace Shapes {
  class Rectangle {
        int length;
        int width;

  public:
        // constructor
        Rectangle ();   //default
        Rectangle (int l, int w);

        // destructor
        virtual ~Rectangle () {};
  public:
        int getArea();
};
```

Classes where & how

- Class definition syntax
  - Usually put in a .h file
  - Members can be methods (functions) or fields (fields)
  - Still include private members here
- Define the member functions
  - Usually in a .cc file
  - (sometimes in line – especially constructors, destructors, getter and setters)
- File names don't have to match the class name

Header file:

```
class Name {
 public:
  // public member definitions & declarations go here

 private:
  // private member definitions & declarations go here
};  // class Name
```

C file:

```
retType Name::MethodName(type1 param1, …, typeN paramN) {
  // body statements
}
```

Class Definition (.h)

- 'const' before arguments specifies that the 'this' object is not changeable
- Inline definitions common for setters and getters
- Google style convection of putting fiels last, with a _ after the name to denote private
- Note still in a header file with header guards (`#ifndef POINT_H`)

```
#ifndef POINT_H_
#define POINT_H_


class Point {
 public:
  Point(const int x, const int y);     // constructor
  int get_x() const { return x_; }     // inline member function
  int get_y() const { return y_; }     // inline member function
  double Distance(const Point& p) const;      // member function
  void SetLocation(const int x, const int y); // member function


 private:
  int x_;  // data member
  int y_;  // data member
};  // class Point


#endif  // POINT_H_
```

Class Member Definition (.cc)

- 'const' is used with an argument to say it shouldn't be modified
  - Use 'const' when applicable as good style ("defensive programming")
- Using 'this' is optional; may need it for name conflicts

```
#include <cmath>
#include "Point.h"

Point::Point(const int x, const int y) {
  x_ = x;
  this->y_ = y;  // "this->" is optional unless name conflicts
}

double Point::Distance(const Point& p) const {
  // We can access p's x_ and y_ variables either through the get_x(),
  // get_y() accessor functions or the x_, y_ private member variables
  // directly, since we're in a member function of the same class.
  double distance = (x_ - p.get_x()) * (x_ - p.get_x());
  distance += (y_ - p.y_) * (y_ - p.y_);
  return sqrt(distance);
}

void Point::SetLocation(const int x, const int y) {
  x_ = x;
  y_ = y;
}
```

Class Usage (.cc)

- Calls constructor to define a new object on the stack
  - But doesn't use 'new' here
- Members are accessed with the dot notation

```
#include <iostream>
#include "Point.h"
using namespace std;

int main(int argc, char** argv) {
  Point p1(1, 2);  // allocate a new Point on the Stack
  Point p2(4, 6);  // allocate a new Point on the Stack


  cout << "p1 is: (" << p1.get_x() << ", ";
  cout << p1.get_y() << ")" << endl;


  cout << "p2 is: (" << p2.get_x() << ", ";
  cout << p2.get_y() << ")" << endl;


  cout << "dist : " << p1.Distance(p2) << endl;
  return 0;
}
```

Stack v. Heap

Java: cannot stack-allocate an object

(only a pointer to one; all objects are dynamically allocated on the heap - all variables are pointers to objects)

`new Thing( ... )` calls constructor, returns heap allocated pointer

C: can stack-allocate a struct, then initialize it (An actual object)

Use malloc and then initialize, must free exactly once later, untyped pointers

C++: stack-allocate and call a constructor Thing t(10000);

Like Java, `new Thing(…)`, but can also do `new int(42)`. Like C must deallocate, but must use delete instead of free. (never mix malloc/free with new/ delete!)

Class Usage with arrays / new (.cc)

- It is possible to define default constructors (next lecture!)
  - The way we declare objects will depend on the available constructors
- We see constructing on the stack (no 'new') and heap (with 'new')
- Notice new/delete array declaration

```
#include "Point.h"

int main() {
  …

  Point stack_pt(1, 2); // stack 2-arg constructor
  Point* heap_pt = new Point(1, 2); // heap 2-arg constructor

  Point* heap_pt_arr_err = new Point[2]; // heap default ctor?
  // fails cause no default ctor

  Point* heap_pt_arr_init_lst = new Point[2]{{1, 2}, {3, 4}};
                                                           // C++11

  …

  delete heap_pt;
  delete[] heap_pt_arr_init_lst;


  return EXIT_SUCCESS;
}
```