What do you think?

Discuss!

Write a list of things you know about C++

- **What is like C?**
- **What is new?**

CSE 374: Lecture 24

C++ Classes

So far …

- Object Oriented
- Larger Language
- Std Library
- Operators
- Namespaces

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;
}
```

…ooks simple enough…

- Compile with **g++** instead of gcc:

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

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`

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

TODAY

‘New’ & ‘Delete’

Const

Classes

Constructing: stack v. heap

New / delete

In C:

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

In C++, we have a nicer syntax for this that does the same thing:

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

New / delete

In C:

```
int
int
fre
fre
```

In C++, w

```
int
int
del
delete [] arr;
```

```
ue 4.
```

‘ new ‘ is an operator, not a function. The operator allocates memory, and then calls a constructor if appropriate.

- Can even initialize primitive data types
- Throws an exception if it fails (not does return NULL)
- Returns memory of the desired type, not an untyped pointer
- Required size calculated by compiler, not by user
- ‘ malloc ‘ does not call a constructor

C++11 `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

new/delete

To allocate on the heap using C++, you use the `new` keyword instead of **`malloc`**`()` from `stdlib.h`

- You can use new to allocate an object (*e.g.* `new Point`)
- You can use new to allocate a primitive type (*e.g.* `new int`)

To deallocate a heap-allocated object or primitive, use the `delete` keyword instead of **`free`**`()` from `stdlib.h`

- 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++

`new/delete` Behavior

`new` behavior:

- When allocating you can specify a constructor or initial value
  - (e.g. `new Point(1, 2)`) or (e.g. `new int(333)`)
- If no initialization specified, it will use default constructor for objects, garbage for primitives (integer, float, character, boolean, double)
  - You don't need to check that `new` returns `nullptr`
  - When an error is encountered, an exception is thrown (that we won't worry about)

`delete` behavior:

- If you `delete` already `deleted` memory, then you will get undefined behavior. (Same as when you double **`free`** in c)

new/delete Example

```
int* AllocateInt(int x) {
  int* heapy_int = new int;
  *heapy_int = x;
  return heapy_int;
}
```

```
Point* AllocatePoint(int x, int y) {
  Point* heapy_pt = new Point(x,y);
  return heapy_pt;
}
```

heappoint.cc

```
#include "Point.h"
using namespace std;

...  // definitions of AllocateInt() and AllocatePoint()

int main() {
  Point* x = AllocatePoint(1, 2);
  int* y = AllocateInt(3);

  cout << "x's x_ coord: " << x->get_x() << endl;
  cout << "y: " << y << ", *y: " << *y << endl;

  delete x;
  delete y;
  return EXIT_SUCCESS;
}
```

Dynamically Allocated Arrays

To dynamically allocate an array:

- Default initialize: `type* name = new type[size];`

To dynamically deallocate an array:

- Use `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

arrays.cc

```
#include "Point.h"

int main() {
  int stack_int;                        // stack (garbage)
  int* heap_int = new int;              // heap (garbage)
  int* heap_int_init = new int(12);     // heap (12)

  int stack_arr[3];                 // stack (garbage)
  int* heap_arr = new int[3];       // heap(garbage)

  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_int;                  // ok
  delete heap_int_init;             // ok
  delete heap_arr;                  // BAD
  delete[] heap_arr_init_val;       // ok

  return EXIT_SUCCESS;
}
```

Arrays Example (primitive)

`malloc` vs. `new`

	`malloc()`	`new`
What is it?	a function	an operator / keyword
How often used (in C)?	often	never
How often used (in C++)?	rarely	often
Allocated memory for	anything	arrays, structs, objects, primitives
Returns	a `void*` (*should be cast*)	appropriate pointer type (*doesn't need a cast*)
When out of memory	returns `NULL`	throws an exception
Deallocating	`free()`	`delete` or `delete[]`

`const` in C++

const

`const`: this cannot be changed/mutated

- Used *much* more in C++ than in C
- Signal of intent to compiler (compile-time errors); meaningless at hardware level

```
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;
}
```

const can be a useful tool for defensive programming

brokenpassbyrefconst.cc

`const` and Pointers

Pointers can change data in two different contexts:

1. You can change the value of the pointer
2. 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*

Example

The syntax with pointers is confusing:

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

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

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

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

  return EXIT_SUCCESS;
}
```

Example

The syntax with pointers is confusing:

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

  const int* z = &y;            // pointer to a (const int)
  *z += 1;                      // compiler error
  z++;                          // ok

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

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

  return EXIT_SUCCESS;
}
```

`const` Parameters

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;
}
```

When to Use References?

Google C++ style guide suggests (not mandated by the C++ language):

- 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;
}
```

styleguide.cc

Questions?

C++ Classes

C Structs: Not object-oriented

```
typedef struct person {
  char* name;
  int age;
} person;
```

```
person *p2;
char name[MAX_NAME];
int  age;
// fill name, age
p2 = makePerson (name, age);
```

```
person* makePerson (char *name, int a) {
  person* p = (person*) malloc (sizeof (person));
  p->name = (char*) malloc (MAX_NAME+1);
  strncpy (p->name, name, MAX_NAME);
  p->age = a;
  return p;
}
```

Notes:
Not self contained
need to allocate heap memory so object will persist
need to allocate memory for the string
Unless you statically declare (char name[MAX_NAME])

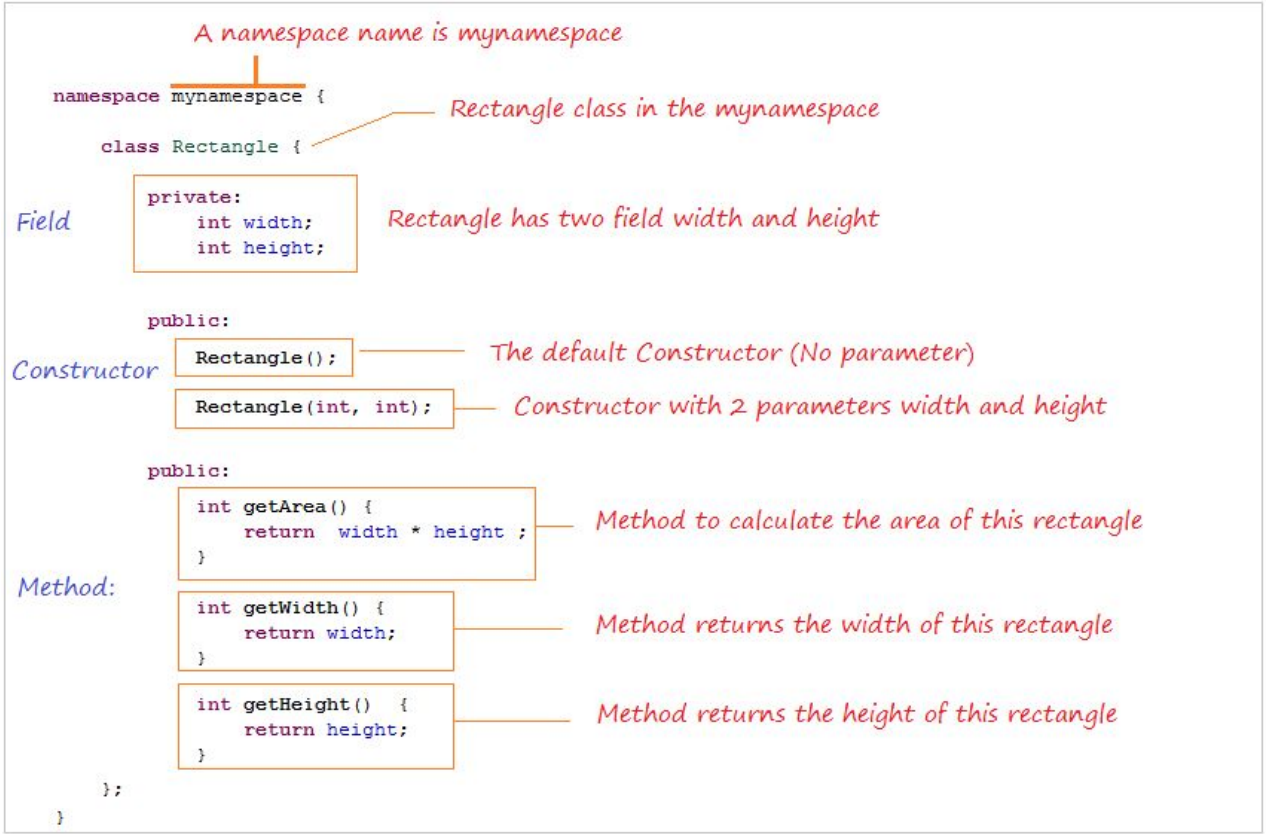
C++ classes: object-oriented

```
class String {
        public:
        String();
        String(const String& other);
        String(const char* raw);
        virtual ~String();
        String& operator=(const String& other);
        size_t length() const;
        void append(const String& other);
        void clear();
        friend std::ostream&
operator<<(std::ostream& out, const String& s);

        private:
        void makeNewRaw(size_t length);
        char* raw_;
};
```

Classes - can define fields and methods

Class layout



Classes

- Like Java
  - Fields vs. methods, static vs. instance, constructors
  - Method overloading (functions, operators, and constructors too)
- Not quite like Java
  - access-modifier (e.g., private) syntax and default
  - declaration separate from implementation (like C)
  - funny constructor syntax, default parameters (e.g., ... = 0)
- Nothing like Java
  - Objects vs. pointers to objects
  - Destructors and copy-constructors
  - virtual vs. non-virtual (to be discussed)

Classes

Class definition syntax (in a `.h` file):

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

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

- Members can be functions (methods) or data (variables)

Class member function definition syntax (in a `.cc` file):

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

- (1) *define* within the class definition or (2) *declare* within the class definition and then *define* elsewhere

Class Organization

It's a little more complex than in C when modularizing with `struct` definition:

- Class definition is part of interface and should go in `.h` file
  - Private members still must be included in definition (**!**)
- Usually put member function definitions into companion `.cc` implementation details
  - Common exception: setter and getter methods
- These files can also include non-member functions that use the class

Why?

Unlike Java, you can name files anything you want

- Typically `Name.cc` and `Name.h` for `class Name`

Class Definition (`.h` file)

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_
```

const means the "this" object we are calling on, can't be changed

Inline definition ok for simple getters/setters

Google C++ naming conventions for data members

Class Member Definitions (. cc file)

Point.cc

```
#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;
}
```

This code uses bad style for demonstration purposes

Can't modify the "this" object inside the function

const won't affect caller, but good style

Can't be const. We have to mutate the Point.

Class Usage (. cc file)

usepoint.cc

```
#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;
}
```

Calls constructor to define an object on the stack (no “new” keyword). More on this shortly.

Dot notation to call function (like Java)

`struct` vs. `class`

In C, a `struct` can only contain data fields

- Has no methods and all fields are always accessible

In C++, `struct` and `class` are (nearly) the same!

- Both define a new type (the `struct` or `class` name)
- Both can have methods and member visibility (public/private/protected)
- Only real (minor) difference: members are default *public* in a `struct` and default *private* in a `class`

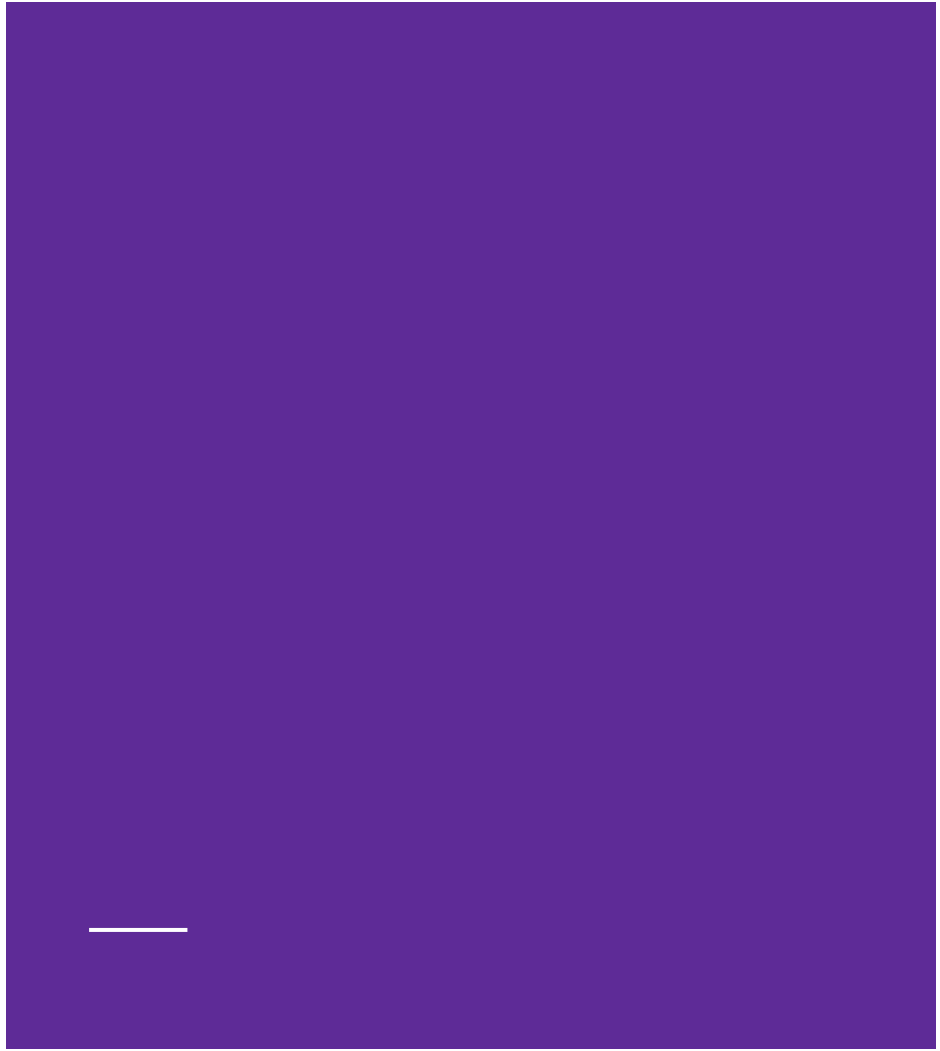
Common style/usage convention:

- Use `struct` for simple bundles of data
- Use `class` for abstractions with data + functions

Questions?

Constructors

ctor



Constructors

A constructor (ctor) initializes a newly-instantiated object

- A class can have multiple constructors that differ in parameters
  - Which one is invoked depends on *how* the object is instantiated

Written with the class name as the method name:

```
Point(const int x, const int y);
```

- C++ will automatically create a synthesized default constructor if you have *no* user-defined constructors
  - Takes no arguments, can be explicitly specified: `Point() = default;`

Synthesized Default Constructor

```
class SimplePoint {
 public:
  // no constructors declared!
  int get_x() const { return x_; }      // inline member function
  int get_y() const { return y_; }      // inline member function
  double Distance(const SimplePoint& p) const;
  void SetLocation(int x, int y);

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

SimplePoint.h

```
#include "SimplePoint.h"
... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
  SimplePoint x;  // invokes synthesized default constructor
  return EXIT_SUCCESS;
}
```

SimplePoint.cc

Synthesized Default Constructor

If you define **any** constructors, C++ assumes you have defined all the ones you intend to be available and will *not* add any others

```
#include "SimplePoint.h"

// defining a constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
  x_ = x;
  y_ = y;
}

void foo() {
  SimplePoint x;          // compiler error: if you define any ctors, C++
                          // will NOT synthesize a default constructor for
                          // you.

  SimplePoint y(1, 2);    // works:invokes the 2-int-arguments constructor
}
```

Multiple Constructors (overloading)

```
#include "SimplePoint.h"

// default constructor
SimplePoint::SimplePoint() {
  x_ = 0;
  y_ = 0;
}

// constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
  x_ = x;
  y_ = y;
}

void foo() {
  SimplePoint x;         // invokes the default constructor
  SimplePoint y(1, 2);   // invokes the 2-int-arguments ctor
  SimplePoint a[3];      // invokes the default ctor 3 times
}
```

Initialization Lists

C++ lets you *optionally* declare an initialization list as part of a constructor definition

- Initializes fields according to parameters in the list
- The following two are (nearly) identical:

```
Point::Point(const int x, const int y) {
  x_ = x;
  y_ = y;
  std::cout << "Point constructed: (" << x_ << ",";
  std::cout << y_<< ")" << std::endl;
}
```

```
// constructor with an initialization list
Point::Point(const int x, const int y) : x_(x), y_(y) {
  std::cout << "Point constructed: (" << x_ << ",";
  std::cout << y_<< ")" << std::endl;
}
```

Body can be empty

Initialization vs. Construction

```
class Point3D {
 public:
  // constructor with 3 int arguments
  Point3D(const int x, const int y, const int z) : y_(y), x_(x) {
    z_ = z;
  }

 private:
  int x_, y_, z_;   // data members
};  // class Point3D
```

First, initialization list is applied.

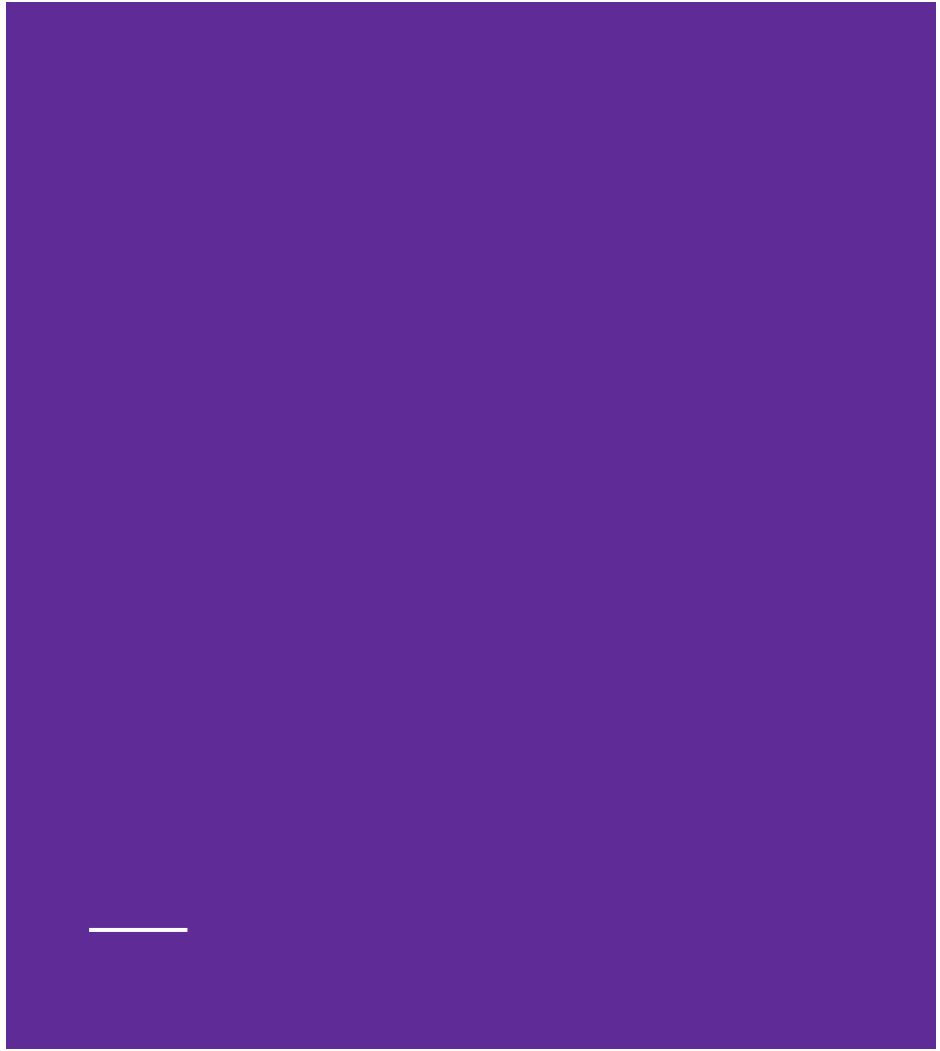
Next, constructor body is executed.

- Data members in initializer list are initialized in the order they are defined in the class, not by the initialization list ordering (**!**)
  - Data members that don't appear in the initialization list are *default initialized/constructed* before body is executed
- Initialization preferred to assignment to avoid extra steps
  - Real code should never mix the two styles

Questions?

Copy Constructors

cctor



Copy Constructors

C++ has the notion of a copy constructor (cctor)

- Used to create a new object as a copy of an existing object

```
Point::Point(const int x, const int y) : x_(x), y_(y) { }

// copy constructor
Point::Point(const Point& copyme) {
  x_ = copyme.x_;
  y_ = copyme.y_;
}

void foo() {
  Point x(1, 2);  // invokes the 2-int-arguments constructor
  Point y(x);     // invokes the copy constructor
  Point z = y;    // also invokes the copy constructor
}
```

Use a cctor since we are constructing based on x

Point z didn't exist before, a ctor must be called

- Initializer lists can also be used in copy constructors (preferred)

Copy Constructors (w/ initialization list)

```
Point::Point(const int x, const int y) : x_(x), y_(y) { }

// copy constructor w/ initialization list
Point::Point(const Point& copyme): x_(copyme.x_), y_(copyme.y_) { }

void foo() {
  Point x(1, 2);  // invokes the 2-int-arguments constructor
  Point y(x);     // invokes the copy constructor
  Point z = y;    // also invokes the copy constructor
}
```

Synthesized Copy Constructor

If you don't define your own copy constructor, C++ will synthesize one for you

- It will do a *shallow* copy of all of the fields (*i.e.* member variables) of your class
  - Does assignment for primitives; could be problematic with pointers
- Sometimes the right thing; sometimes the wrong thing

```
#include "SimplePoint.h"                    // In this example, synthesized cctor is fine

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
  SimplePoint x;
  SimplePoint y(x);  // invokes synthesized copy constructor
  ...
  return EXIT_SUCCESS;
}
```

When Do Copies Happen?

The copy constructor is invoked if:

- You *initialize* an object from another object of the same type:

```
Point x;        // default ctor
Point y(x);     // copy ctor
Point z = y;    // copy ctor
```

- You pass a non-reference object as a **value** parameter to a function:

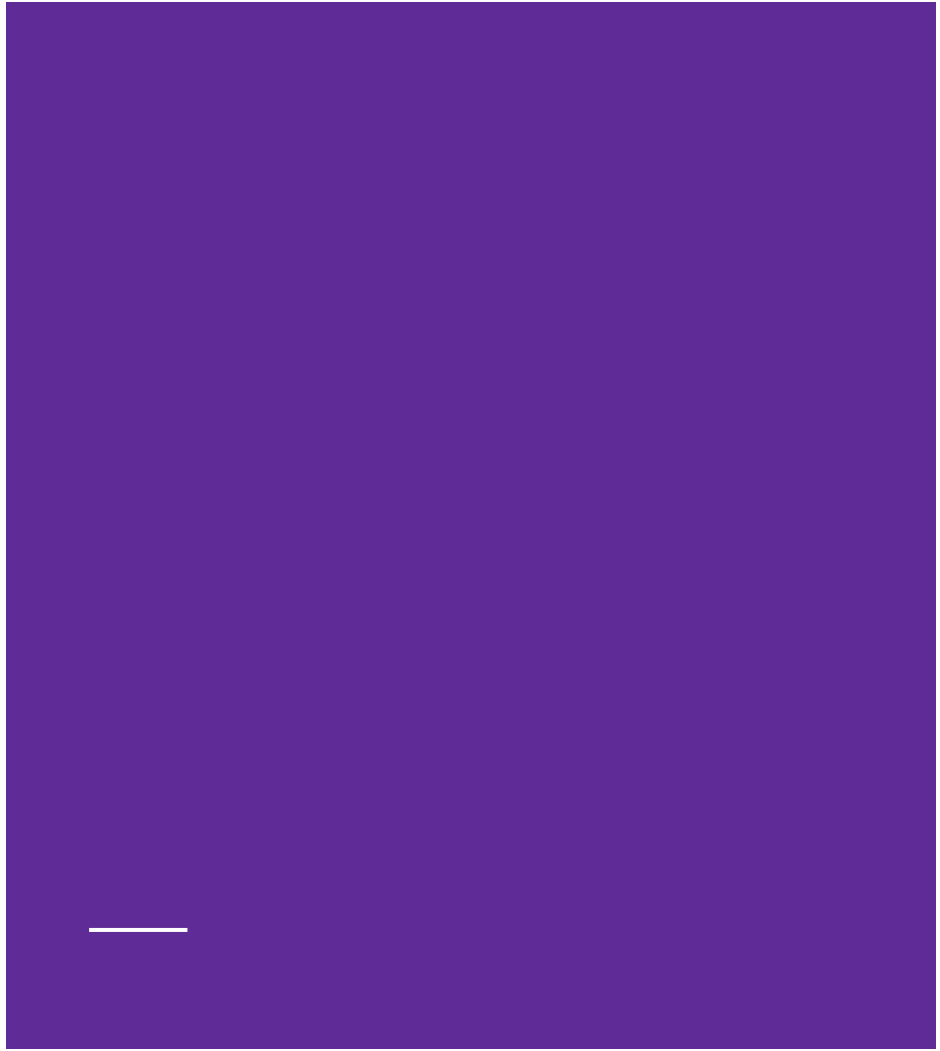
```
void foo(Point x) { ... }

Point y;        // default ctor
foo(y);         // copy ctor
```

- You return a non-reference object **value** from a function:

```
Point foo() {
  Point y;      // default ctor
  return y;     // copy ctor
}
```

Destructors



Destructors

C++ has the notion of a destructor (dtor)

- Invoked automatically when a class instance is deleted (even via exceptions or other causes!)
- Place to put your cleanup code – free any dynamic storage or other resources owned by the object
- Standard C++ idiom for managing dynamic resources
  - Slogan: "*Resource Acquisition Is Initialization*" (RAII)

```
Point::~Point() {   // destructor
  // do any cleanup needed when a Point object goes away
  // (nothing to do here since we have no dynamic resources)
}
```

Destructor Example

```
class FileDescriptor {
  public:
    FileDescriptor(char * file) { // Constructor
      fd_ = open(file, O_RDONLY);
      // Error checking omitted
    }
    ~FileDescriptor() { close(fd_); } // Destructor
    int get_fd() const { return fd_; } // inline member function

  private:
    int fd_; // data member
}; // class FileDescriptor
```

Without destructor, the file wouldn't be closed

FileDescriptor.h

```
#include "FileDescriptor.h"
int main(int argc, char** argv) {
  FileDescriptor fd(foo.txt);
  return EXIT_SUCCESS;
}
```

Destruct the object when it falls out of scope (when we return)

Implicit constructors & destructors

Conversion constructors are implicit: automatically applied when a constructor is called with one argument.

If you want a single argument constructor that is not implicit, must use

```
explicit String(const
char* raw);
```

Destructors are used by ‘delete’ to clean up when freeing memory.

```
Virtual ~String();
```

You do not call destructors explicitly

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)

C: can stack-allocate a struct, then initialize it (An actual object)
C++: stack-allocate and call a constructor (where this is the object's address, as always, except this is a pointer) Thing t(10000);

Java: new Thing(...) calls constructor, returns heap allocated pointer
C: Use malloc and then initialized, must free exactly once later, untyped pointers
C++: 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!)

arrays.cc

```
#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;
}
```

Arrays Example (class objects)