C++ Classes, Constructors and Copies (oh my!)

CSE 333 Autumn 2025

Instructors: Naomi Alterman, Chris Thachuk

Teaching Assistants:

Ann Baturytski	Derek de Leuw	Blake Diaz
Rishabh Jain	Chendur Jel Jayavelu	Lucas Kwan
Irene Xin Jie Lau	Nathan Li	Maya Odenheim
Advay Patil	Selim Saridede	Deeksha Vatwani
Angela Wu	Jiexiao Xu	

Vibe check

- PollEv.com/naomila

- How are you feeling about C++ right about now?

Administrivia

- Naomi's office hours moving to Wednesdays 2:30-3:30p
- No exercise 9 yet (out on Monday)
- A bunch of deadlines coming up in the next few weeks, plan ahead!!!
  - EX9 (it's a big one), EX10, HW2, midterm

Lecture Outline

- **Classes**
- Constructors
- Copy Constructors
- Assignment
- Destructors
- An extended example

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

- 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` file with implementation details
  - These files can also include non-member functions that use the class
  - Common exception: setter and getter methods

- Unlike Java, you can name files anything you want
  - Typically `Name.cc` and `Name.h` for `class Name`

Big ol' example

Point.h

```
#ifndef POINT_H_
#define POINT_H_

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

 private:
  int x_;
  int y_;
};

#endif  // POINT_H_
```

Point.cc

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

// NOTE: THIS FILE HAS WEIRD AND BAD CODING STYLE!
// We just want to illustrate some language features for you (:

Point::Point(const int x, const int y) {
  x_ = x;
  this->y_ = y;
}

double Point::Distance(const Point& p) const {
  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;
}
```

Const & Classes

- Like other data types, **objects** can be declared as `const`:
  - Once a `const` object has been constructed, its member variables can't be changed
  - Can only invoke member functions that are labeled `const`

- You can declare a member **function** of a class as `const`
  - This means that if cannot modify the object it was called on
    - The compiler will treat member variables as `const` inside the function at compile time
  - If a member function doesn't modify the object, mark it `const`!

Class Usage (. cc file)

usepoint.cc

```
#include <iostream>
#include <cstdlib>
#include "Point.h"

using namespace std;

int main(int argc, char** argv) {
  // allocate two Points on the Stack, call the default ctor for each
  Point p1(1, 2);
  Point p2(4, 6);
  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 EXIT_SUCCESS;
}
```

Lecture Outline

- Classes
- **Constructors**
- Copy Constructors
- Assignment
- Destructors
- An extended example

Constructors

- A constructor (ctor) initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameter count and type
  - Written with the class name as the method name and **no return type** (**!**) :

```
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
  - Calls the default ctors on all non-"plain old data" (non-POD) member variables
  - Leaves the "plain old data" uninitialized (**!**)
  - Will fail if you have non-initialized const or reference data members

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(const int x, const 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 0;
}
```

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 a[3];      // invokes the default ctor 3 times
                         //   (fails if no default ctor)
  SimplePoint y(1, 2);   // invokes the 2-int-arguments ctor
}
```

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;
}
```

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 of default initialization (construction) followed by assignment
- (and no, real code should never mix the two styles this way ☺)

Lecture Outline

- Classes
- Constructors
- **Copy Constructors**
- Assignment
- Destructors
- An extended example

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 p1(1, 2);  // invokes the 2-int-arguments constructor

  Point p2(p1);    // invokes the copy constructor
                   // could also be written as "Point p2 = p1;"
}
```

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

When Do Copies Happen?

- The copy constructor is invoked if:
  - You *initialize* an object from another object of the same type:

```
Point p1;       // default ctor
Point p2(p1);    // copy ctor
Point p3 = p2;   // copy ctor
```

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

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

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

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

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

Compiler Optimization

- The compiler sometimes uses a "return by value optimization" or "move semantics" to eliminate unnecessary copies
  - Sometimes you might not see a constructor get invoked when you might expect it

```
Point foo() {
  Point pt;          // default ctor
  return pt;         // copy ctor? optimized?
}

Point pt1(1, 2);     // two-ints-argument ctor
Point pt2 = pt1;     // copy ctor
Point pt3 = foo();   // copy ctor? optimized?
```

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
  - Sometimes the right thing; sometimes the wrong thing

```
#include "SimplePoint.h"

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

int main(int argc, char** argv) {
  SimplePoint pt1;
  SimplePoint pt2(pt1);  // invokes synthesized copy constructor
  ...
  return 0;
}
```

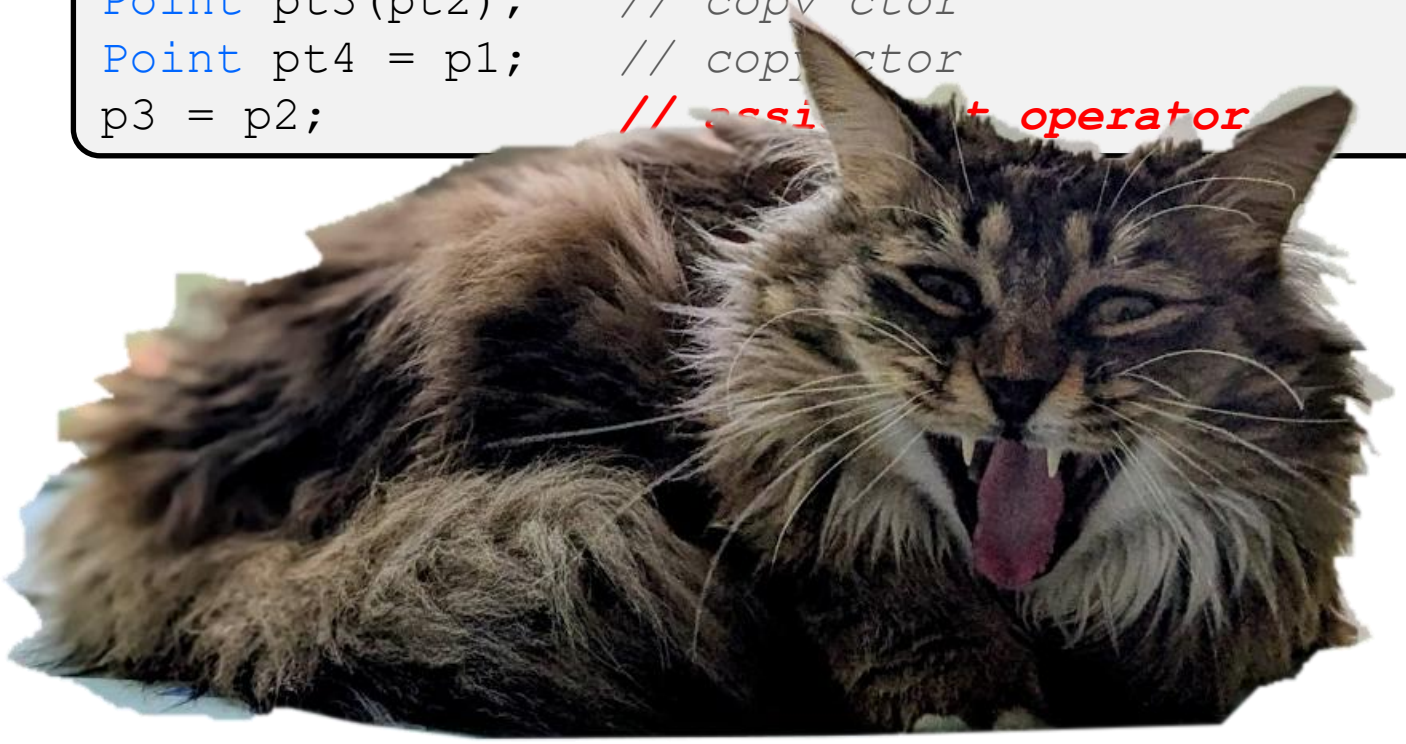
Lecture Outline

- Classes
- Constructors
- Copy Constructors
- **Assignment**
- Destructors
- An extended example

Assignment ain't Construction

- "=" is the assignment operator
  - Assigns values to an *existing, already constructed* object

```
Point pt1;          // default ctor
Point pt2(1, 2);    // two-ints-argument ctor
Point pt3(pt2);     // copy ctor
Point pt4 = p1;     // copy ctor
p3 = p2;            // assignment operator
```



Assignment ain't Construction

- "=" is the assignment operator
  - Assigns values to an *existing, already constructed* object

```
Point pt1;          // default ctor
Point pt2(1, 2);    // two-ints-argument ctor
Point pt3(pt2);     // copy ctor
Point pt4 = p1;     // copy ctor
p3 = p2;            // assignment operator
```

  - How can you tell the difference between assignment `operator=` and a copy constructor that uses =?
    - Answer: are you creating/initializing a new object? If so, it's a copy constructor; if you are just updating an existing object it's assignment

Overloading the "=" Operator

- You can choose to define the "=" operator
  - But there are some rules you should follow:

```
Point& Point::operator=(const Point& rhs) {
  if (this != &rhs) {  // (1) always check against this
    x_ = rhs.x_;
    y_ = rhs.y_;
  }
  return *this;        // (2) always return *this from op=
}

Point c;         // default constructor
a = b = c;       // works because = return *this
a = (b = c);     // equiv. to above (= is right-associative)
(a = b) = c;     // "works" because = returns a non-const
```

Synthesized Assignment Operator

- If you don't define the assignment operator, C++ will synthesize one for you
  - It will do a *shallow* copy of all of the fields (*i.e.* member variables) of your class
  - Sometimes the right thing; sometimes the wrong thing

```
#include "SimplePoint.h"

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

int main(int argc, char** argv) {
  SimplePoint pt1;
  SimplePoint pt2(pt1);
  pt2 = pt1;            // invokes synthesized assignment operator
  return 0;
}
```

Lecture Outline

- Classes
- Constructors
- Copy Constructors
- Assignment
- **Destructors**
- An extended example

Destructors

- C++ has the notion of a destructor (dtor)
  - Invoked automatically when a class instance is deleted, goes out of scope, etc. (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)
}
```

Lecture Outline

- Classes
- Constructors
- Copy Constructors
- Assignment
- Destructors
- **An extended example**

Complex Example Walkthrough

See:

`Complex.h`

`Complex.cc`

`testcomplex.cc`

- (Some details like friend functions and namespaces are explained in more detail next lecture, but ideas should make sense from looking at the code and explanations in *C++ Primer*.)

Extra Exercise #1

- Modify your Point3D class from Lec 10 Extra #1
  - Disable the copy constructor and assignment operator
  - Attempt to use copy & assignment in code and see what error the compiler generates
  - Write a `CopyFrom()` member function and try using it instead
    - (See details about `CopyFrom()` in next lecture)

Extra Exercise #2

- Write a C++ class that:
  - Is given the name of a file as a constructor argument
  - Has a `GetNextWord()` method that returns the next whitespace- or newline-separated word from the file as a copy of a `string` object, or an empty string once you hit EOF
  - Has a destructor that cleans up anything that needs cleaning up