Poll Everywhere

pollev.com/cse333

Vibe Check: How are you feeling about C++ right now?

- You can keep this poll open and change your answer as the lecture goes on!

Systems Programming

C++ Classes, Constructors, and Copies

Instructor: Justin Hsia

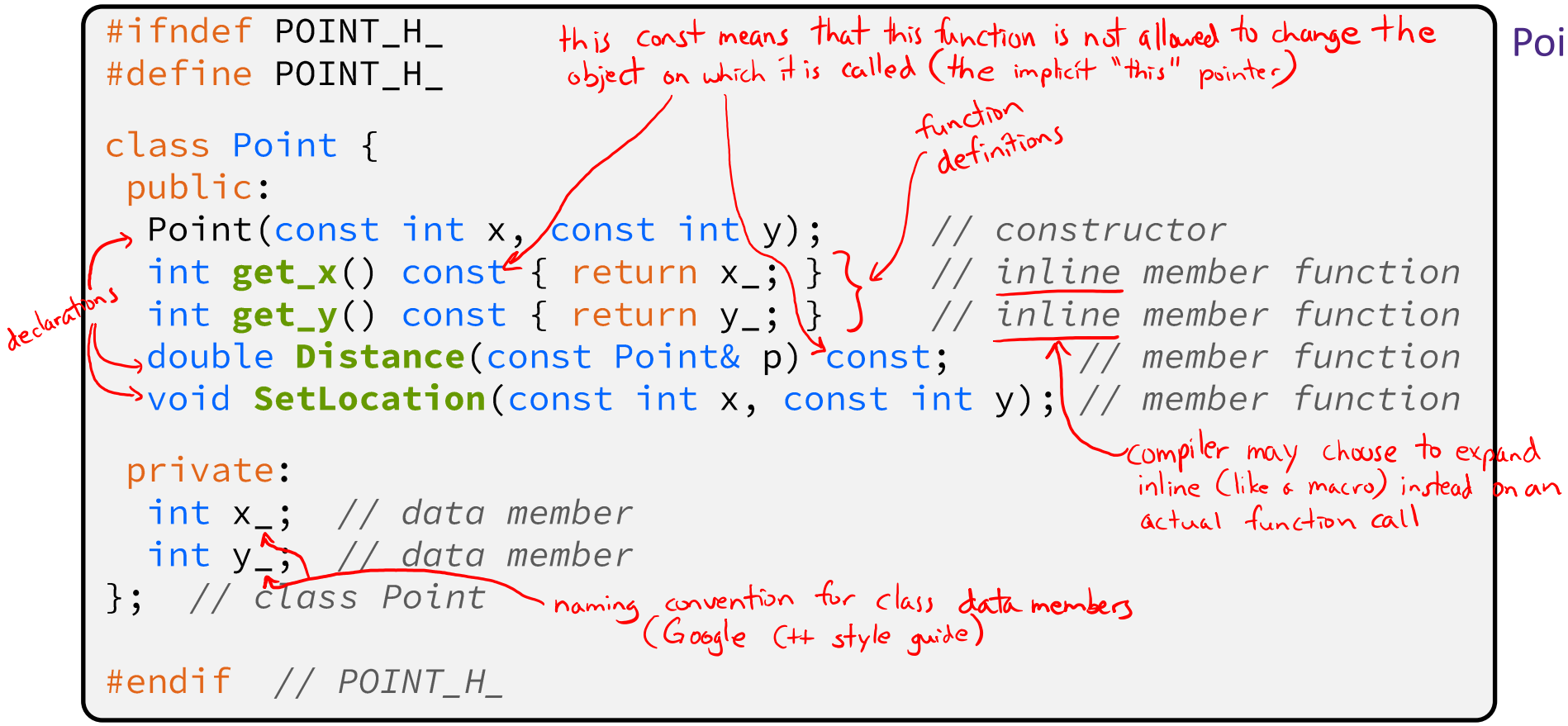
Teaching Assistants:

Aadi Adgaonkar	Advaith Vijayakumar	Alice Fu
Andrew Vlahovich	Aviel Wood	Camden Harris
Esha Gore	Hannah Hempstead	Helena Wang
Laura Wang	Megan Frisella	Nina Tharamal
Shrihun Sankepally	Shriya Sateesh	Yusong Yan

Relevant Course Information

- Exercise 6 released today, due Wednesday
  - Write a class in C++ (today's topic!) and Makefile
- Homework 2 due this coming Thursday (10/29)
  - File system crawler, indexer, and search engine
  - "Check-In 1 (HW2)" activity due Tuesday (10/27)
- Midterm is Monday, November 2 from 5:30–6:40 PM
  - Located in GUG 220 or ARC 147, depending on your quiz section
  - Ed post with details forthcoming
  - Practice midterms and solutions on the course website
  - Midterm review (hybrid) on Friday (10/30) from 4:30–6:20 PM

Class Definition (.h file)



Point.h

Class Member Definitions (.cc file)

Point.cc

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

Point::Point(const int x, const int y) {
  // BAD STYLE to mix-and-match "this->var_" and "var_"
  x_ = x;
  this->y_ = y;  // "this->" is optional unless name conflicts
}

double Point::Distance(const Point& p) const {
  // In a member function of the same class, can access p's x_
  // and y_ members directly or using the get_x()/get_y()
  // accessor functions (BAD STYLE to mix-and-match, though).
  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` file)

usepoint.cc

```
#include "Point.h"

#include <iostream>
#include <cstdlib>

using std::cout;
using std::endl;

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 EXIT_SUCCESS;
}
```

calls defined constructor

"dot notation" used for member functions

`struct` vs. `class`

- In C, a `struct` can only contain data fields
  - No methods and all fields are always accessible

- In C++, `struct` and `class` are (nearly) the same!
  - Both can have methods and member visibility (public/private/protected)
  - Minor difference: members are default *public* in a `struct` and default *private* in a `class`

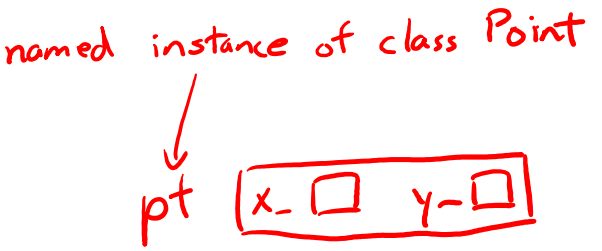
- Common style convention:
  - Use `struct` for simple bundles of data ← public data members with names like x, y
  - Use `class` for abstractions with data + functions ↑ private data members with names like x_, y_

Memory Diagrams for Objects

- An **object** is an instance of a class that maintains its *state* independently of other objects
  - This state is the collection of its data members
  - Conceptually, an object acts like a collection of data fields (plus class metadata)
    - Layout is *not* specified or guaranteed, unlike structs in C

- Drawn out as variables within variables:

```
class Point {
 ...

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



Lecture Outline (1/4)

- **Constructors**
- Copy Constructors
- Assignment
- Destructors

Constructors

- A constructor (ctor) initializes a newly-instantiated object
  - Written with the class name as the method name:
    e.g., `Point(const int x, const int y);`
  - A class can have multiple constructors that differ in parameters
  - A constructor *must* be invoked when creating a new instance of an object – which one depends on *how* the object is instantiated
- C++ will automatically create a synthesized default constructor if you have *no* user-defined constructors
  - Takes no arguments and calls the default ctor on all non-"plain old data" (non-POD) member variables
  - Synthesized default ctor will fail if you have non-initialized const or reference data members

created for you

zero-argument

Synthesized Default Constructor Example

SimplePoint.h

```
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
```

default behavior → primitives: just allocate space (mystery data)
default behavior → objects: default construct

SimplePoint.cc

```
#include "SimplePoint.h"

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

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

(main) x [x_ ? y_ ?]

Synthesized Default Constructor

- If you define *any* constructors, C++ will *not* add a synthesized default constructor
  - This doesn't prevent it from synthesizing other methods (*e.g.*, copy constructor)

```
#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
}
```

added, so no synthesized def ctor

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
}
```

added, so now there is a def. ctor

int: a [? | ? | ?]

SimplePoint: a [x_[0] y_[0] | x_[0] y_[0] | x_[0] y_[0]]

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;
}
```

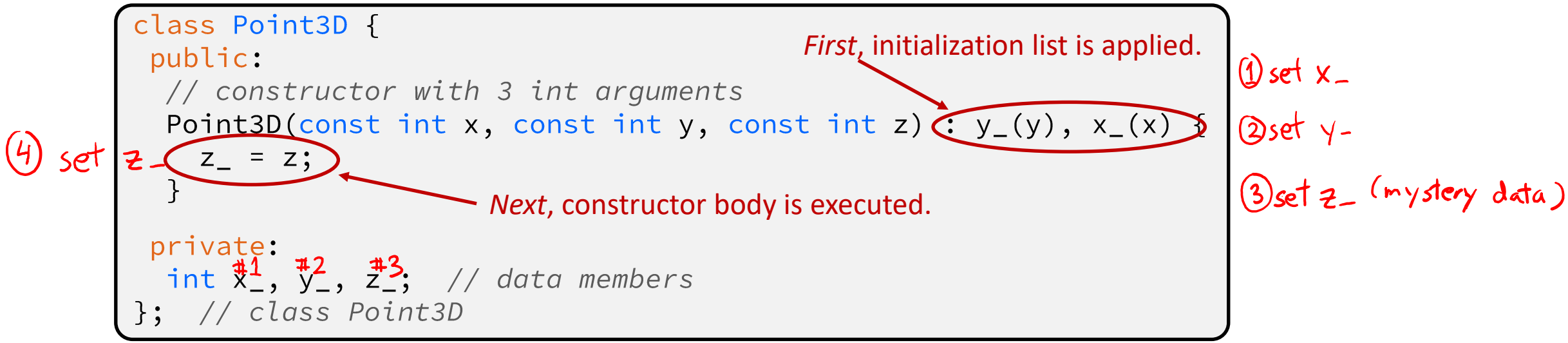
```
// constructor with an initialization list
Point::Point(const int x, const int y) : x_(x), y_(y) {
  std::cout << "Point constructed: (" << x_ << ",";
  std::cout << y_<< ")" << std::endl;
}
```

body can be empty { }

can be expressions

member names

Initialization vs. Construction



- 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

Lecture Outline (2/4)

- Constructors
- **Copy Constructors**
- Assignment
- Destructors

Copy Constructors

- C++ has the notion of a copy constructor (cctor)
  - Used to initialize a new object "as a copy" of an existing object
  - Initializer lists can also be used in copy constructors (preferred)

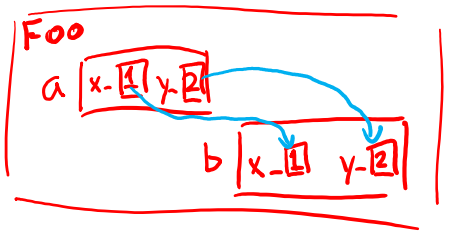
```
Point::Point(const int x, const int y) : x_(x), y_(y) { }

// copy constructor
Point::Point(const Point& orig) x_(orig.x_), y_(orig.y_) { }

void Foo() {
  Point a(1, 2); // invokes the 2-int-arguments constructor

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

reference to object of same class

alias binds to object

constructing from existing object, so we use the copy ctor

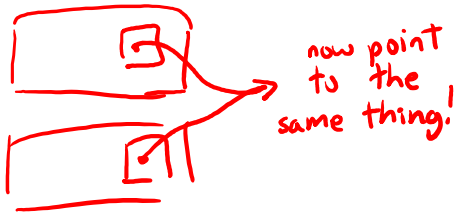
a ctor must be called because the object didn't exist previously



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 (can be problematic with pointers)
  - Sometimes the right thing; sometimes the wrong thing
  - This is separate from non-copy constructors

```
#include "SimplePoint.h"

... // 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
```

pass-by-value of an object

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

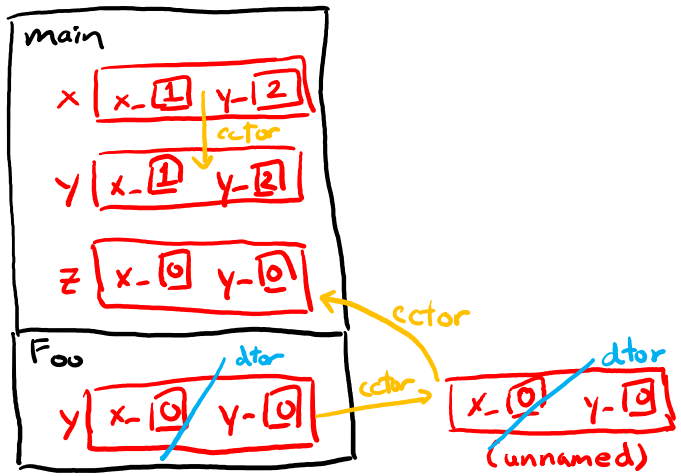
```
Point Foo() {
  Point y;     // default ctor
  return y;    // copy ctor
}
```

Compiler Optimization: "Copy Elision"

"Eliminate" → "elide" → "elision"

- The compiler may eliminate unnecessary copies
  - You might not see a constructor get invoked when you expect it
- Most common is when an object is returned by value (*i.e.*, copied) and passed into *another* copy constructor
  - Since C++17, this kind of copy elision is **guaranteed**

```
Point Foo() {
  Point y;          // default ctor
  return y;         // copy ctor? optimized?
}

int main(int argc, char* argv[]) {
  Point x(1, 2);    // two-ints-argument ctor
  Point y = x;      // copy ctor
  Point z = Foo();  // copy ctor? optimized?
}
```



Lecture Outline (3/4)

- Constructors
- Copy Constructors
- **Assignment**
- Destructors

Assignment Isn't Construction

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

```
Point w;          // default ctor
Point x(1, 2);    // two-ints-argument ctor
Point y(x);       // copy ctor
Point z = w;      // copy ctor (z did not exist)
y = x;            // assignment operator (y exists)
```

method operator=( )

Overloading the "=" Operator

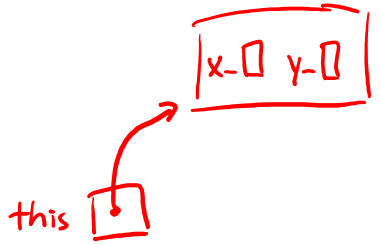
- You can choose to define the "=" operator
  - But there are some rules you should follow:

```
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 a;          // 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
```

more important when dealing with dynamically allocated memory

returns reference to class object (allows for chaining)

a.operator=(b.operator=(c))



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

Usually wrong whenever class owns a resource (e.g., dynamically allocated data)

```
#include "SimplePoint.h"

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

int main(int argc, char* argv[]) {
  SimplePoint x;
  SimplePoint y(x);
  y = x;           // invokes synthesized assignment operator
  return EXIT_SUCCESS;
}
```

Lecture Outline (4/4)

- Constructors
- Copy Constructors
- Assignment
- **Destructors**

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 for your cleanup code – free any resources owned by the object
    - Standard C++ idiom for managing dynamic resources: "*Resource Acquisition Is Initialization*" (**RAII**)
  - After destructor body finishes, destruct members in reverse order of declaration (*i.e.*, reverse of initialization list)

tilde — no parameters

```
Point::~Point() {   // destructor
  // Do any cleanup needed when a Point object goes away.
  // Nothing to do here, but what if we had dynamic resources?
}
```

executed in reverse order as ctor: ① body of dtor
② destruct members in reverse order of declaration

Destructor Example

FileDescriptor.h

```
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
```

dtor automatically closes file for the user!

use_fd.cc

```
#include "FileDescriptor.h"

int main(int argc, char* argv[]) {
  FileDescriptor fd1("foo.txt");
  return EXIT_SUCCESS;
}
```

destruct object when it falls out of scope (here, when we return)

Poll Everywhere

pollev.com/cse333

How many times does the *destructor* get invoked?

- Assume `Point` with ctor, cctor, =, dtor, and member function
  `double Distance(const Point& p) const;`
- Assume no optimizations

ctor	cctor	op=	dtor

A. 1 B. 2 C. 3 D. 4

E. We're lost…

poll.cc

```
Point PrintRad(Point& pt) {
  Point origin(0, 0);
  double r = origin.Distance(pt);
  double theta = atan2(pt.get_y(), pt.get_x());
  cout << "r = " << r << endl;
  cout << "theta = " << theta << " rad" << endl;
  return pt;
}

int main(int argc, char* argv[]) {
  Point pt(3, 4);
  PrintRad(pt);
  return EXIT_SUCCESS;
}
```

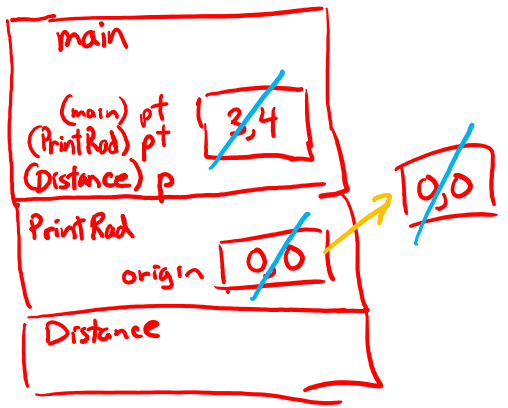
Polling Solution

ctor	cctor	op=	dtor
2	1	0	3

- How many times does the ***destructor*** get invoked?
  - Assume `Point` with ctor, cctor, =, dtor, and member function `double Distance(const Point& p) const;`
  - Assume no optimizations

poll.cc

```
Point PrintRad(Point& pt) {
  Point origin(0, 0);          // ② ctor called
  double r = origin.Distance(pt);   // Distance takes ref, so object NOT copied
  double theta = atan2(pt.get_y(), pt.get_x());
  cout << "r = " << r << endl;
  cout << "theta = " << theta << " rad" << endl;
  return pt;                   // ③ PrintRad returns an object, so cctor is called to create a temp
}                              // ④ while cleaning up, origin is destructed

int main(int argc, char* argv[]) {
  Point pt(3, 4);              // ① ctor called
  PrintRad(pt);                // PrintRad takes ref, so pt is NOT copied
  return EXIT_SUCCESS;         // ⑤ return value of PrintRad ignored; temp is destructed
}                              // ⑥ while cleaning up, pt is destructed
```



Extra Exercise #1

- Write a C++ program that:
  - Has a class representing a 3-dimensional point
  - Has the following methods:
    - Return the inner product of two 3D points
    - Return the distance between two 3D points
    - Accessors and mutators for the `x`, `y`, and `z` coordinates

Extra Exercise #2

- Write a C++ program that:
  - Has a class representing a 3-dimensional box
    - Use your Extra Exercise #1 class to store the coordinates of the vertices that define the box
    - Assume the box has right-angles only and its faces are parallel to the axes, so you only need 2 vertices to define it
  - Has the following methods:
    - Test if one box is inside another box
    - Return the volume of a box
    - Handles `<<`, `=`, and a copy constructor
    - Uses `const` in all the right places

Extra Exercise #3

- 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