

C++ Constructor Insanity

CSE 333 Summer 2020

Instructor: Travis McGaha

Teaching Assistants:

Jeter Arellano

Ian Hsiao

Ramya Challa

Allen Jung

Kyrie Dowling

Sylvia Wang



pollev.com/cse33320su

About how long did Exercise 9 take?

- A. 0-1 Hours
- B. 1-2 Hours
- C. 2-3 Hours
- D. 3-4 Hours
- E. 4+ Hours
- F. I didn't submit / I prefer not to say

Side question:

What's a good question to ask
students before lecture? :thinking:

Administrivia

- ❖ Exercise 10 released today, due Monday
 - Write a substantive class in C++!
 - Uses a lot of what we will talk about in lecture
- ❖ Homework 2 due Thursday (7/23)
 - File system crawler, indexer, and search engine
 - Note: libhw1.a (yours or ours) and the .h files from hw1 need to be in right directory (~yourgit/hw1/)
 - Note: use Ctrl-D to exit searchshell, test on directory of small self-made files
- ❖ Mid-Quarter Survey. Out later today, due in a week (7/24)

Class Definition (.h file)



Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {           const means the object
public:                 we are calling on, can't be changed   Inline definition ok for simple
    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
```

Declarations

Google C++ naming conventions for data members

```
#endif // POINT_H_
```

Class Member Definitions (.cc file)

Point.cc

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

This code uses bad style for demonstration purposes

```
Point::Point(const int x, const int y) {
    x_ = x;           Equivalent to y_=y;
    this->y_ = y;    // "this->" is optional unless name conflicts
}                 "this" is a Point* const
double Point::Distance(const Point& p) const {Can't modify the "this" object inside the function
    // 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);
}                 const won't affect caller, but good style
void Point::SetLocation(const int x, const int y) {Can't be const. We have to mutate the Point
    x_ = x;
    y_ = y;
}
```

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)

Dot notation to call function
(like java)

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-*

Lecture Outline

- ❖ Constructors
- ❖ Copy Constructors
- ❖ Assignment
- ❖ Destructors

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
 - A constructor is always invoked when creating a new instance of an object.
- ❖ Written with the class name as the method name:

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

zero arg

- C++ will automatically create a synthesized default constructor if you have *no* user-defined constructors
Created for you
 - 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

Synthesized Default Constructor Example

```
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 initializes fields:

- If primitive, garbage values (like normal vars)
- If object, run default ctor

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
    SimplePoint y(1, 2);     // ctors, C++ will NOT synthesize a
                           // default constructor for you.

    // works: invokes the 2-int-arguments
    // constructor
}
```

Because we defined
a ctor already

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
}
```

Constructs points with default ctor. { (0,0), (0,0), (0,0) }

Note if we used primitives instead of objects, the primitives will contain garbage bytes

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

data member name

Expression



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; // 1) set x_  
    } // 2) set y_  
    private:  
        int x_, y_, z_; // data members  
}; // class Point3D
```

First, initialization list is applied.

Next, constructor body is executed.

Annotations:

- Red circle around `z_ = z;` labeled *4) set z_*
- Red circle around `y_(y), x_(x)` labeled *1) set x_* and *2) set y_*
- Red circle around `z_` labeled *3) set z_ (garbage)*

- 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

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



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  
                  // Use a cctor since we are constructing based on x  
    Point y(x);   // invokes the copy constructor  
                  // could also be written as "Point y = x;"  
}
```

Reference to object of same type

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

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

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
 - ↑ *Calls ctor of data members that are objects*
 - Does assignment for primitives*
 - Could be problematic with pointers*
 - Sometimes the right thing; sometimes the wrong thing

```
#include "SimplePoint.h"                                // In this example, synthesized ctor 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
}
```

Compiler Optimization

- ❖ The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies *Read on own if interested. Briefly discussed in future lecture*
 - Sometimes you might not see a constructor get invoked when you might expect it

```
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?  
}
```

Compiler Optimization

Note: Arrow points to *next* instruction.

- ❖ The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies *Read on own if interested. Briefly discussed in future lecture*
 - Sometimes you might not see a constructor get invoked when you might expect it

```
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?  
}
```



Compiler Optimization

Note: Arrow points to *next* instruction.

- ❖ The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies *Read on own if interested. Briefly discussed in future lecture*
 - Sometimes you might not see a constructor get invoked when you might expect it

main stack frame

x

{1, 2}

```
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?  
}
```



Compiler Optimization

Note: Arrow points to *next* instruction.

- ❖ The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies *Read on own if interested. Briefly discussed in future lecture*
 - Sometimes you might not see a constructor get invoked when you might expect it

main stack frame

x	{1, 2}
y	{1, 2}

```
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?  
}
```



Compiler Optimization

Note: Arrow points to *next* instruction.

- ❖ The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies *Read on own if interested. Briefly discussed in future lecture*
 - Sometimes you might not see a constructor get invoked when you might expect it

main stack frame

x	{1, 2}
y	{1, 2}

foo stack frame



```
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?  
}
```

Compiler Optimization

Note: Arrow points to *next* instruction.

- ❖ The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies Read on own if interested. Briefly discussed in future lecture
- Sometimes you might not see a constructor get invoked when you might expect it

main stack frame

x	{1, 2}
y	{1, 2}

foo stack frame

y	{0, 0}
---	--------



```
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?  
}
```

Compiler Optimization

Note: Arrow points to *next* instruction.

- ❖ The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies *Read on own if interested. Briefly discussed in future lecture*
 - Sometimes you might not see a constructor get invoked when you might expect it

main stack frame

x	{1, 2}
y	{1, 2}

foo stack frame

y	{0, 0}
---	--------

?? Temp object ??

temp	{0, 0}
------	--------

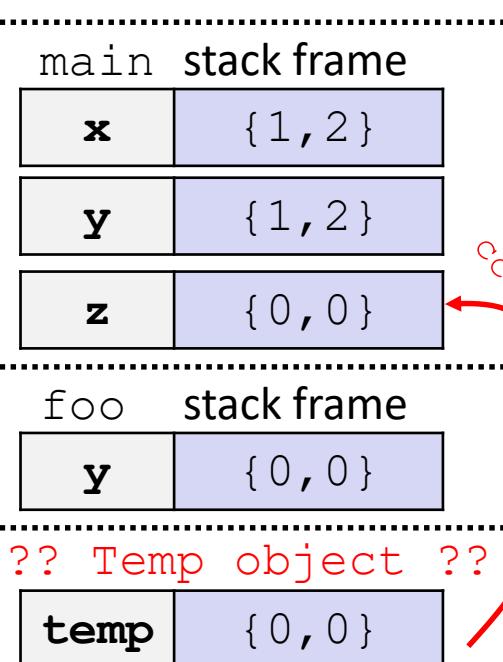
```
Point foo() {  
    Point y;           // default ctor  
    return y;          // copy ctor? optimized?  
}  
  
int main(int argc, char** argv) {  
    Point x(1, 2);    // two-ints-argument ctor  
    Point y = x;       // copy ctor  
    Point z = foo();   // copy ctor? optimized?  
}
```



Compiler Optimization

Note: Arrow points to *next* instruction.

- ❖ The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies *Read on own if interested. Briefly discussed in future lecture*
 - Sometimes you might not see a constructor get invoked when you might expect it



```
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

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

Assignment != 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
y = x;           // assignment operator
```

 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=  
} // Should be a reference  
// to *this to allow chaining
```

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
// reference to *this

Explicit equivalent:

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 a class has 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

- ❖ 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 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)

tilde No parameters

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

When a destructor is invoked:

1. run destructor body
2. Call destructor of any data members

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
```

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)
```



Poll Everywhere

pollev.com/cse33320su

- ❖ How many times does the **destructor** get invoked?
 - Assume Point with everything defined (ctor, cctor, =, dtor)
 - Assume no compiler optimizations

Trace through entire code! See if you can also count ctor, cctor & op=

A. 1

B. 2

C. 3

D. 4

E. We're lost...

test.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 0;  
}
```



Poll Everywhere

pollev.com/cse33320su

- ❖ How many times does the **destructor** get invoked?
 - Assume Point with everything defined (ctor, cctor, =, dtor)
 - Assume no compiler optimizations
- Note: Arrow points to *next instruction*. test.cc

main

```
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 0;  
}
```

ctor	cctor	Op=	dtor
0	0	0	0



Poll Everywhere

pollev.com/cse33320su

- ❖ How many times does the **destructor** get invoked?

- Assume Point with everything defined (ctor, cctor, =, dtor)
- Assume no compiler optimizations

Note: Arrow points
to *next* instruction. test.cc

main

pt {3, 4}

```
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 0;
}
```



ctor	cctor	Op=	dtor
1	0	0	0



Poll Everywhere

pollev.com/cse33320su

- ❖ How many times does the **destructor** get invoked?

- Assume Point with everything defined (ctor, cctor, =, dtor)
- Assume no compiler optimizations

Note: Arrow points
to *next instruction*. test.cc

main

pt(main)	{ 3, 4 }
pt(PrintRad)	

PrintRad

```
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 0;  
}
```

ctor	cctor	Op=	dtor
1	0	0	0



Poll Everywhere

pollev.com/cse33320su

- ❖ How many times does the **destructor** get invoked?

- Assume Point with everything defined (ctor, cctor, =, dtor)
- Assume no compiler optimizations

Note: Arrow points
to *next* instruction. test.cc

main

pt(main)	{ 3, 4 }
pt(PrintRad)	

PrintRad

origin	{ 0, 0 }
--------	----------

```
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 0;
}
```

ctor	cctor	Op=	dtor
2	0	0	0



Poll Everywhere

pollev.com/cse33320su

- ❖ How many times does the **destructor** get invoked?

- Assume Point with everything defined (ctor, cctor, =, dtor)
- Assume no compiler optimizations

Note: Arrow points
to *next* instruction. test.cc

main

pt(main)	{ 3, 4 }
pt(Print Rad)	

PrintRad

origin	{ 0, 0 }
--------	----------

Point::Distance

// Takes a const
// ref, just
// computation

```
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 0;
}
```

ctor	cctor	Op=	dtor
2	0	0	0



Poll Everywhere

pollev.com/cse33320su

- ❖ How many times does the **destructor** get invoked?

- Assume Point with everything defined (ctor, cctor, =, dtor)
- Assume no compiler optimizations

Note: Arrow points
to *next instruction.* test.cc

main

pt(main)	{ 3, 4 }
pt(Print Rad)	

PrintRad

origin	{ 0, 0 }
--------	----------

?? Temp object ??

temp	{ 3, 4 }
------	----------

```
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 0;
}
```

ctor	cctor	Op=	dtor
2	1	0	0


pollev.com/cse33320su

❖ How many times does the ***destructor*** get invoked?

- Assume Point with everything defined (ctor, cctor, =, dtor)
- Assume no compiler optimizations

Note: Arrow points
to *next instruction*. test.cc

main

pt(main)	{ 3, 4 }
pt(PrintRad)	

PrintRad

origin	{ 0, 0 }
--------	----------

```
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 0;
}
```

?? Temp object ??

temp	{ 3, 4 }
------	----------

ctor	cctor	Op=	dtor
2	1	0	1



Poll Everywhere

pollev.com/cse33320su

- ❖ How many times does the **destructor** get invoked?

- Assume Point with everything defined (ctor, cctor, =, dtor)
- Assume no compiler optimizations

Note: Arrow points
to *next instruction.* test.cc

main

pt

{ 3, 4 }

```
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 0;  
}
```



?? Temp object ??

temp

{ 3, 4 }

ctor	cctor	Op=	dtor
2	1	0	2



Poll Everywhere

pollev.com/cse33320su

- ❖ How many times does the **destructor** get invoked?
 - Assume Point with everything defined (ctor, cctor, =, dtor)
 - Assume no compiler optimizations

Note: Arrow points
to *next* instruction. test.cc

main

pt

{3, 4}

```
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 0;  
}
```

C. 3



ctor	cctor	Op=	dtor
2	1	0	3

Monday Preview

```
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
```

```
#include "FileDescriptor.h"
```

```
int main(int argc, char** argv) {  
    FileDescriptor fd1("foo.txt");  
    FileDescriptor fd2(fd1); // Invokes synthesized ctor??  
    return EXIT_SUCCESS; // What happens when we return  
} // and destruct our objects?
```

Synthesized ctor just copies the data members (fd_)

What happens when we return and destruct our objects?

(This won't crash the program, but what if we were using heap allocation instead of file descriptors?)

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