Copying Objects

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat
Jialin Cai
Mati Davis
Donglun He
Chandravaran Kunjeti
Heyi Liu
Shufan Liu
Eddy Yang

Poll Everywhere

pollev.com/tqm

- How was spring break?

Logistics

- Check-in 06 Due Friday 3/17 @ 10 am
  - Releases later tonight
- Mid Semester Survey Due Monday 3/20 @ 11:59 pm
  - Graded, but all submissions get 100% credit
- Project details & HW3 to be released soon-ish
  - Project to be done in pairs
- Grades
  - Midterm grading is being worked on
  - HW2 grades are in progress
    still missing submissions from about 1/4th of the class

Lecture Outline

- **Review**
  - **References**
  - **Classes, Ctor, Dtor**
- Copy Constructor
- Assignment Operator
- Move

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);       // 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_
```

Declarations

Poll Everywhere

pollev.com/tqm

- Final output of this code?

```
int& stuff(int& x, int y) {
  int& z = y;
  z = 12;
  x += 3;
  return x;
}

int main() {
  int a = 1;
  int b = 2;

  int& c = stuff(a, b);
  c++;

  cout << a << endl;
  cout << b << endl;
  cout << c << endl;
}
```

Poll Everywhere

pollev.com/tqm

- How many times does a **`string`** constructor get invoked here?

```
int main() {
  string a("hello");
  string b("like");
  string* c = new string("antennas");
}
```

Poll Everywhere

pollev.com/tqm

- How many times does the **`string`** destructor get invoked here?

```
int main() {
  string a("hello");
  string b("like");
  string* c = new string("antennas");
}
```

Lecture Outline

- Review
  - References
  - Classes, Ctor, Dtor
- **Copy Constructor**
- Assignment Operator
- Move

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
                  // could also be written as "Point y = x;"
}
```

Reference to object of same type

Use a cctor since we are constructing based on x

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 cctor 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 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
}
```

Compiler Optimization

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

`main` stack frame

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

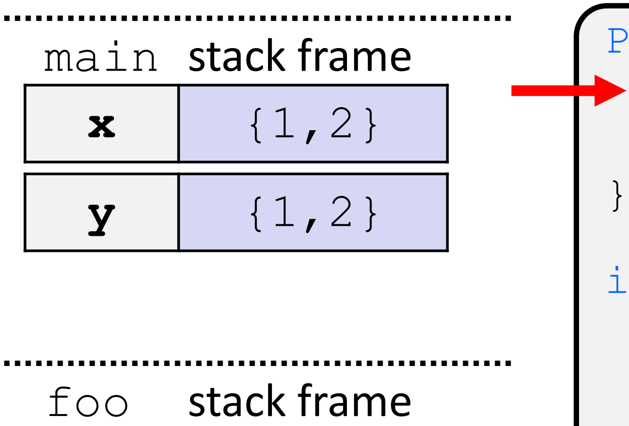
```
Point foo() {
  Point y;            // default ctor
  return y;           // copy ctor? optimized?
}

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

<u>Note</u>: Arrow points to *next* instruction.

- The compiler sometimes uses a "return by value optimization" or "<u>move semantics</u>" to eliminate unnecessary copies *Briefly discussed later in lecture*
  - Sometimes you might not see a constructor get invoked when you might expect it



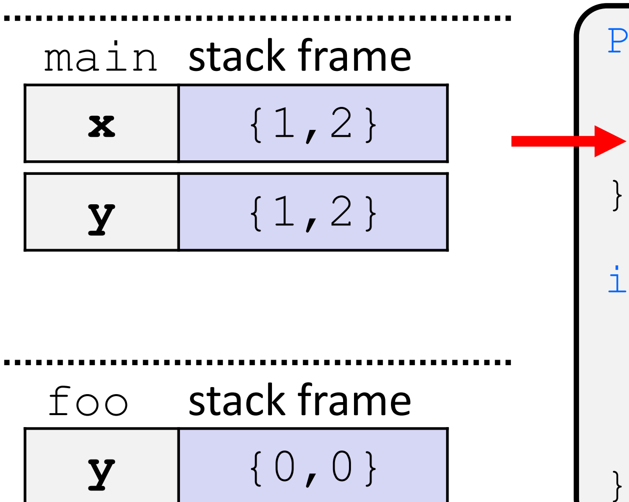

```
Point foo() {
  Point y;          // default ctor
  return y;         // copy ctor? optimized?
}

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 Briefly discussed later in lecture
  - Sometimes you might not see a constructor get invoked when you might expect it



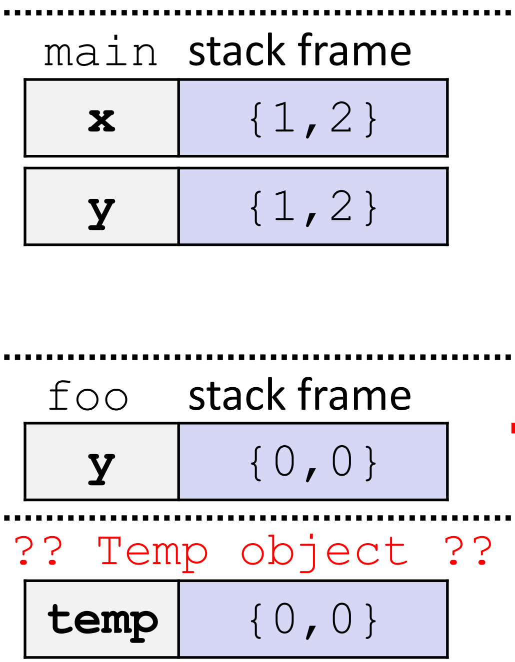

```
Point foo() {
  Point y;         // default ctor
  return y;        // copy ctor? optimized?
}

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 Briefly discussed later in lecture
  - Sometimes you might not see a constructor get invoked when you might expect it



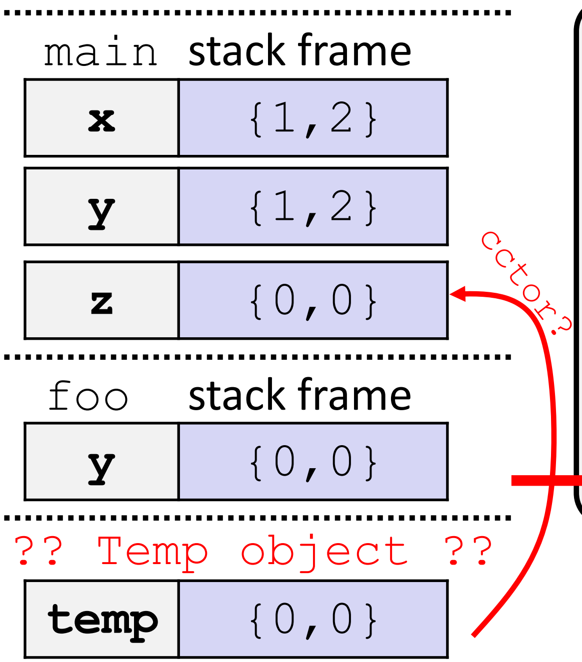

```
Point foo() {
  Point y;         // default ctor
  return y;        // copy ctor? optimized?
}

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 *Briefly discussed later in 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

- Review
  - References
  - Classes, Ctor, Dtor
- Copy Constructor
- **Assignment Operator**
- Move

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=()`

equivalent code:
`y.operator=(x);`

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
                 // reference to *this
```

More important when data members are Dynamic memory

Should be a reference to *this to allow chaining

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;
}
```

Poll Everywhere

pollev.com/tqm

- 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=

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;
}
```

A. 1
B. 2
C. 3
D. 4
E. We’re lost…

Poll Everywhere

pollev.com/tqm

- 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/tqm

- 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;
}
```

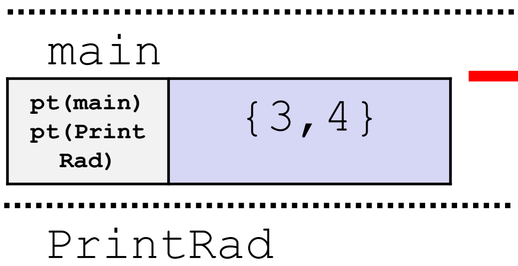
ctor	cctor	Op=	dtor
1	0	0	0

Poll Everywhere

pollev.com/tqm

- 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

```
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;
}
```

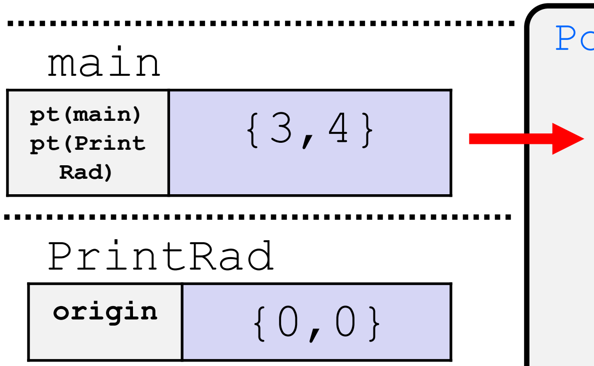
ctor	cctor	Op=	dtor
1	0	0	0

Poll Everywhere

pollev.com/tqm

- 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

```
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;
}
```

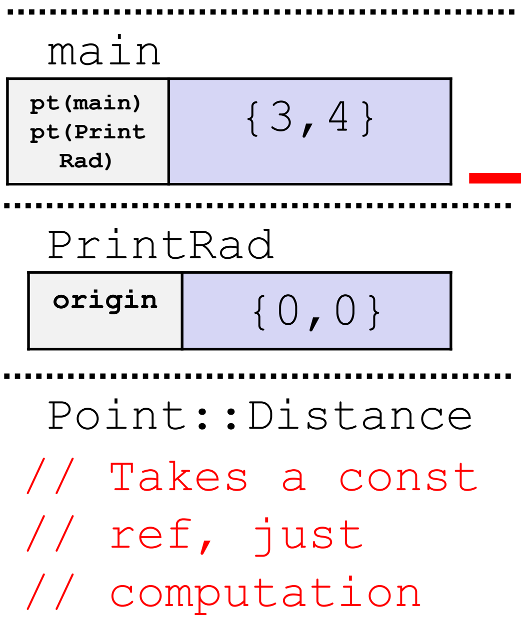
ctor	cctor	Op=	dtor
2	0	0	0

Poll Everywhere

pollev.com/tqm

- 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

```
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/tqm

- 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.

main

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

PrintRad

origin	{0,0}

?? Temp object ??

temp	{3,4}

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;   // <- arrow
}

int main(int argc, char** argv) {
  Point pt(3, 4);
  PrintRad(pt);
  return 0;
}
```

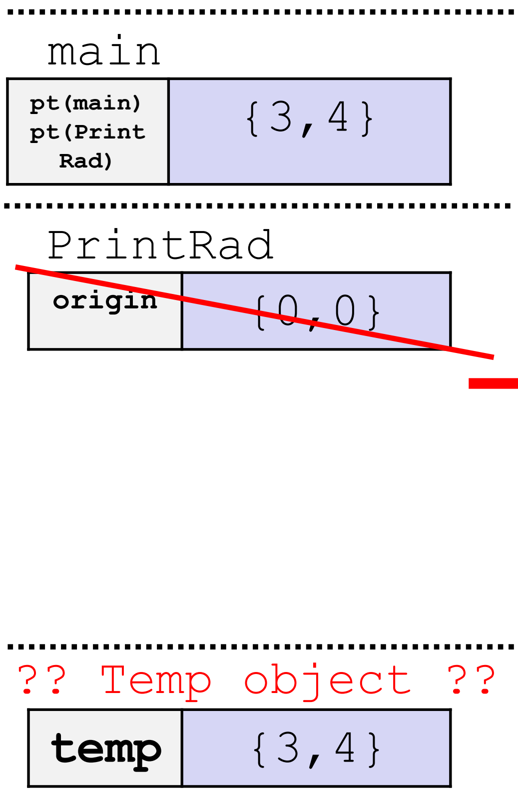
ctor	cctor	Op=	dtor
2	1	0	0

Poll Everywhere

pollev.com/tqm

- 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

```
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	1

Poll Everywhere

pollev.com/tqm

- 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

```
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;
}
```

main

pt	{3,4}

?? Temp object ??

temp	{3,4}

ctor	cctor	Op=	dtor
2	1	0	2

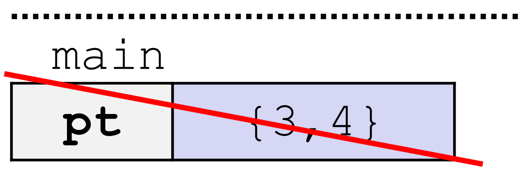
Poll Everywhere

pollev.com/tqm

- 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




```
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

Lecture Outline

- Review
  - References
  - Classes, Ctor, Dtor
- Copy Constructor
- Assignment Operator
- **Move**

Move Semantics (C++11)

- "Move semantics" move values from one object to another without copying ("stealing")
  - A complex topic that uses things called "*rvalue references*"
    - Mostly beyond the scope of this class

```
std::string ReturnString(void) {
  std::string x("Quack");
  // this return might copy
  return x;
}

int main(int argc, char **argv) {
  std::string a("bleg");

  // moves a to b
  std::string b = std::move(a);
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;

  // moves the returned value into b
  b = std::move(ReturnString());
  std::cout << "b: " << b << std::endl;

  return EXIT_SUCCESS;
}
```

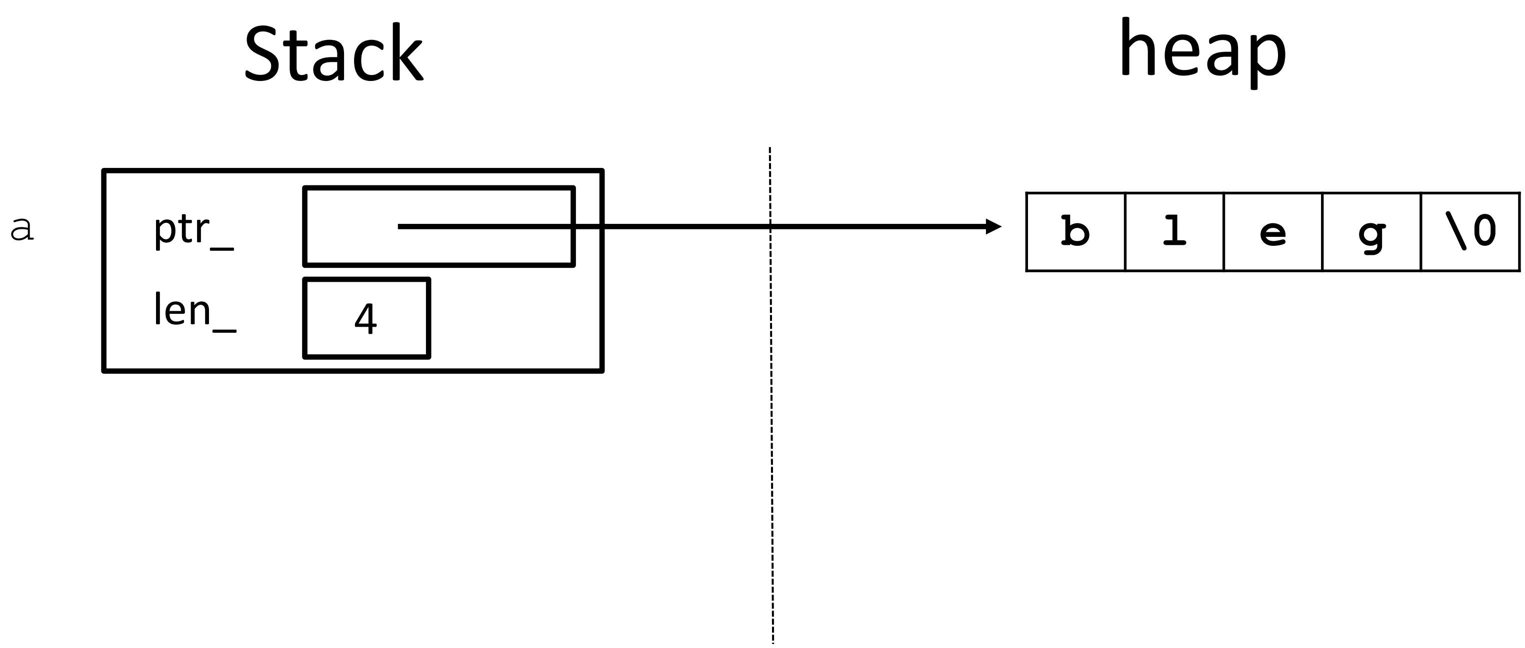
a: ""
b: "bleg"

Move Semantics: close up look

- Internally a string manages a heap allocated C string and looks something like:

```
int main(int argc, char **argv) {
  std::string a("bleg");


}
```



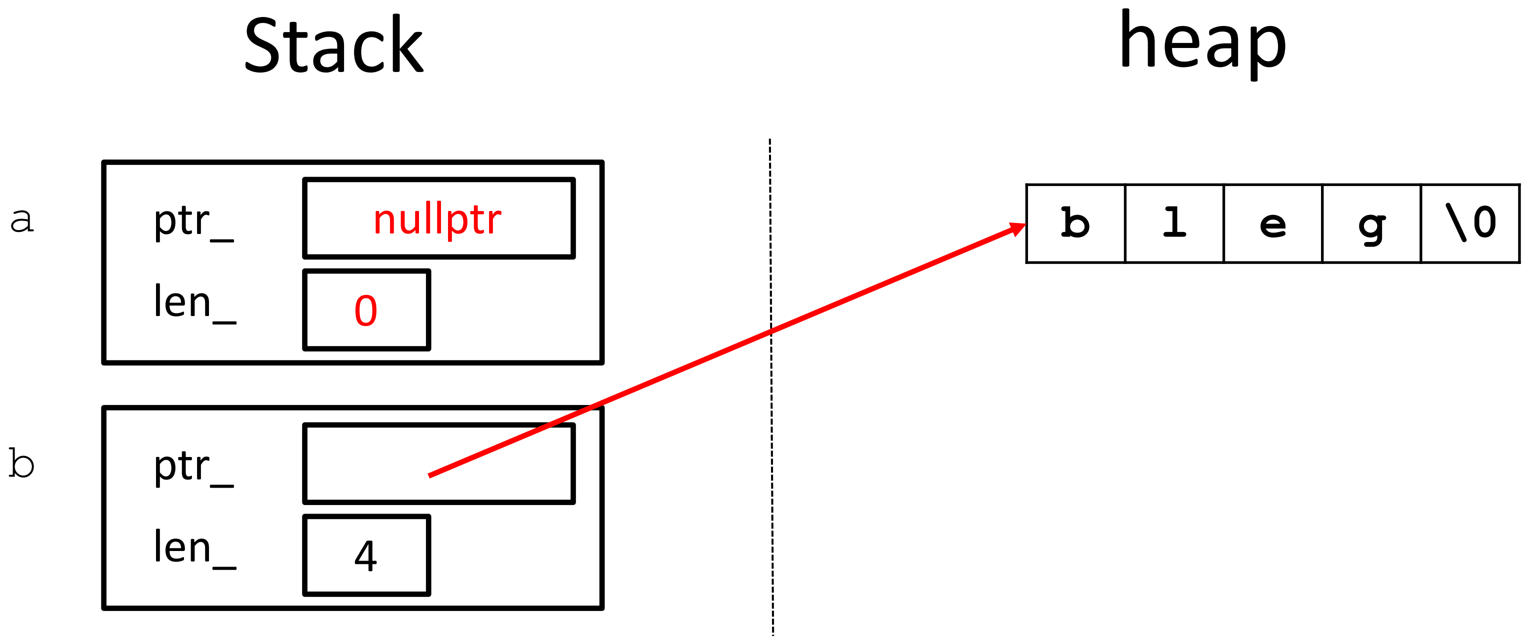
Move Semantics: close up look

- When we use move to construct string **b**

```
int main(int argc, char **argv) {
  std::string a("bleg");

  std::string b = std::move(a);
}
```

we could get something like:



Move Semantics: Use Cases

- Useful for optimizing away temporary copies
- Preferred in cases where copying may be expensive
- Can be used to help enforce uniqueness

Move Semantics: Details

- Implement a "Move Constructor" with something like:

```
Point::Point(Point&& other) {
  // ...
}
```

- Implement a "Move assignment" with something like:

```
Point& Point::operator=(Point&& rhs) {
  // ...
}
```

Move Semantics: Details

- "Move Constructor" example for a fake **`String`** class:

```
String::String(String&& other) {
  this->len_ = other.len_;
  this->ptr_ = other.ptr_;

  other.len_ = 0;
  other.ptr_ = nullptr;
}
```

std::move

- Use `std::move` to indicate that you want to move something and not copy it

```
Point p (3, 2);              // constructor
Point a (p);                 // copy constructor

Point b (std::move(p)); // move constructor
```