Object Copying & Casts

Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

Ash Fujiyama
CV Kunjeti
Felix Sun
Heyi Liu
Kevin Bernat
Lang Qin
Sean Chuang
Serena Chen
Yuna Shao

Logistics

- Exam grades to be posted today
  - Regrade requests open 24 hours after grades are posted
  - Will be open for a week
  - Rember that we have the clobber policy, it is ok if the exam did not go well.

- HW03 to be released today: due Friday next week
- Project to be posted soon
  - Partner sign up released today, due on Friday

- Checkin Due before Lecture on Wednesday
  - Released last week

Poll Everywhere

pollev.com/tqm

- Any questions?

Lecture Outline

- **Review**
  - **References**
  - **Classes, Ctor, Dtor**
  - **New/delete**
- Copy Constructor
- Assignment Operator
- Casting

Class Definition (.hpp file)

Point.hpp

```
#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");
}
```

`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 string("hi")`)
  - If no initialization specified, it will use default constructor for objects, garbage for primitives More on constructors in Wednesday's lecture
  - 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 `delete`d 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;
}
```

```
string* AllocateStr(char* str) {
  string* heapy_str = new string(str);
  return heapy_str;
}
```

heappoint.cc

```
#include "Point.h"

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

int main() {
  string* x = AllocateStr("Hello 595!");
  int* y = AllocateInt(3);

  cout << "x's value: " << *x << endl;
  cout << "y: " << y << ", *y: " << *y << endl;

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

Dynamically Allocated Arrays

new still returns a pointer of specified type

- 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 Example (primitive)

arrays.cpp

```
#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
                                              // 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;
}
```

Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

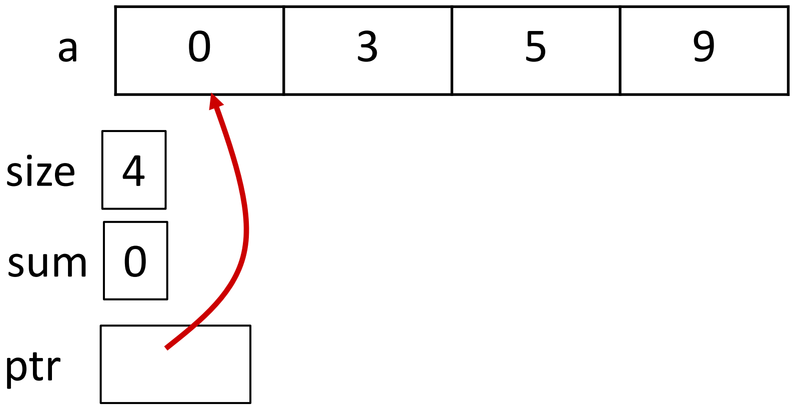
```
int* a = new int[4]{0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
for (int i = 0; i < size; i++) {
  sum += ptr[i];
}
```

```
int* a = new int[4]{0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```

Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

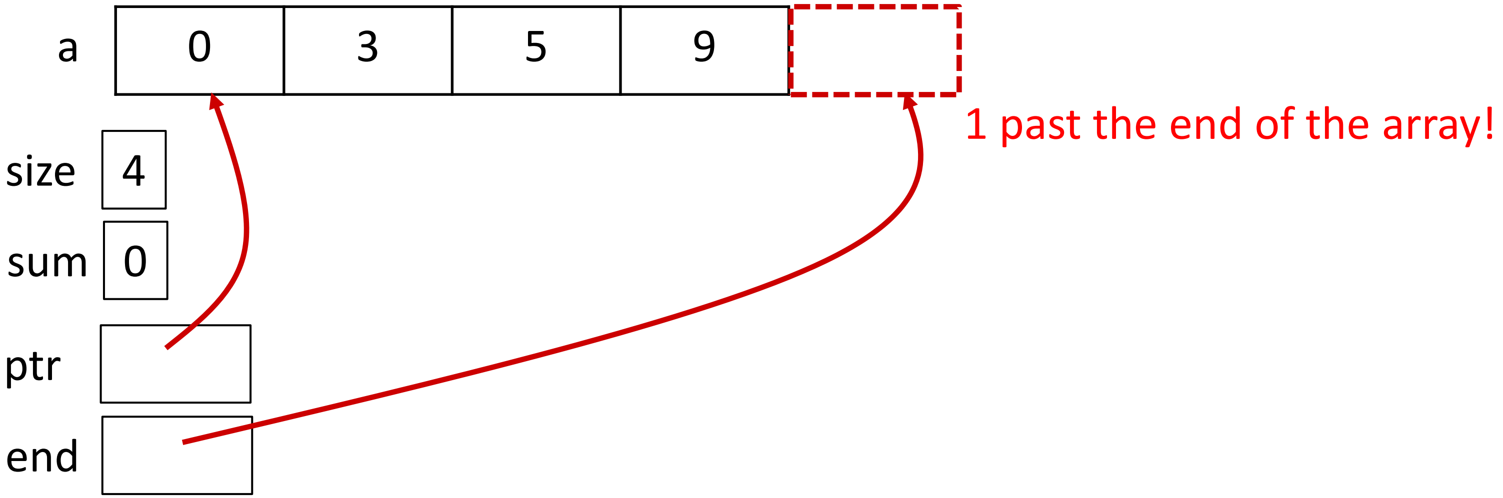
```
int* a = new int[4]{0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

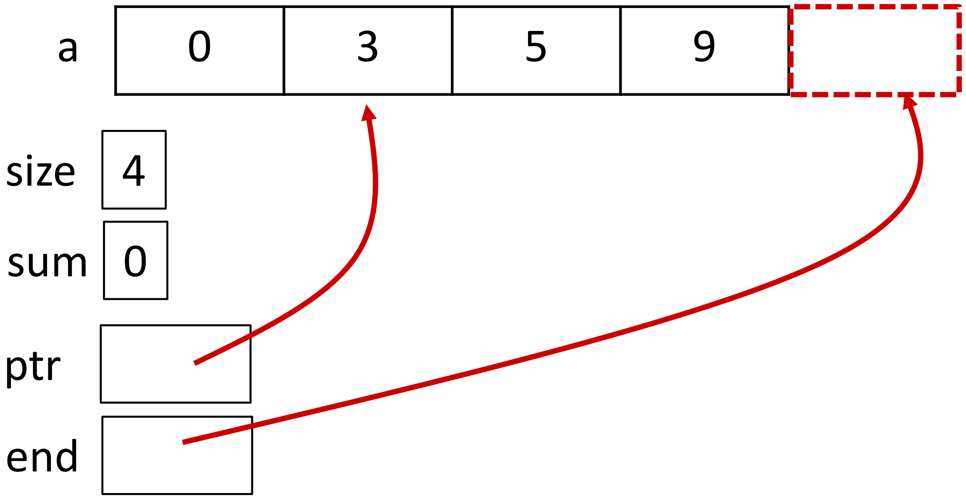
```
int* a = new int[4]{0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

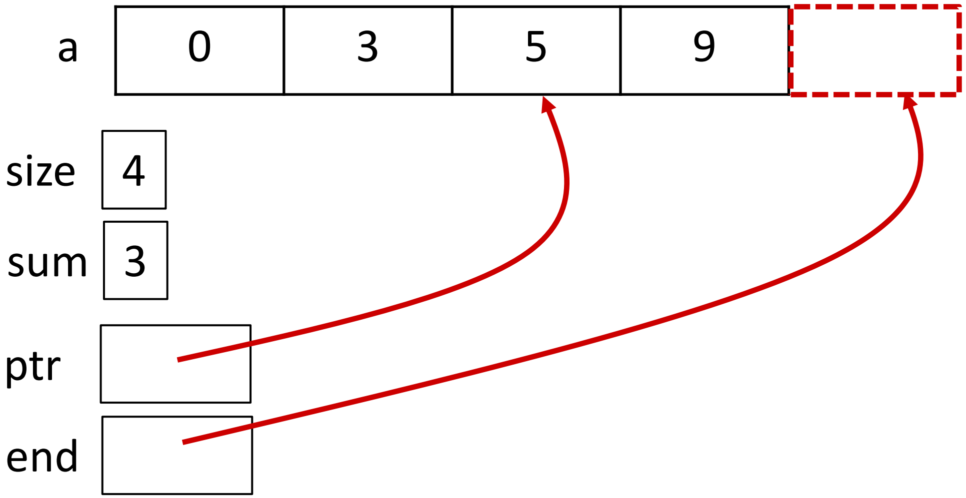
```
int* a = new int[4]{0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

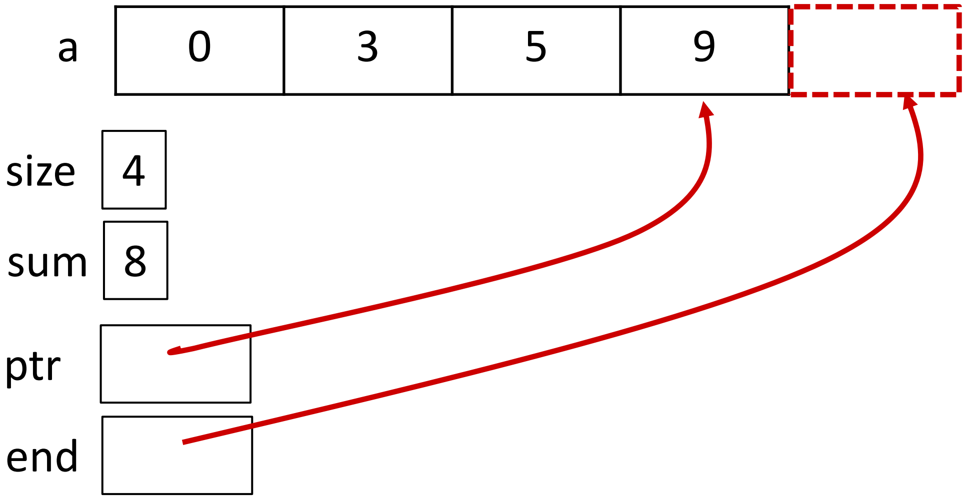
```
int* a = new int[4]{0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

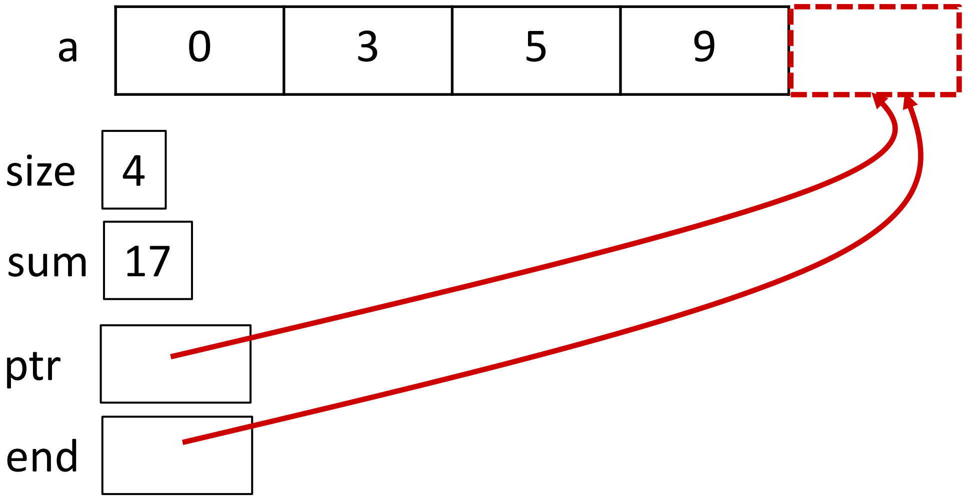
```
int* a = new int[4]{0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

```
int* a = new int[4]{0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Lecture Outline

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

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

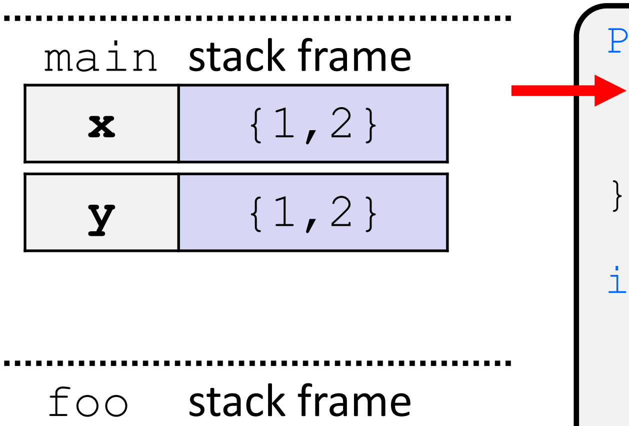
x	{1,2}
y	{1,2}

```
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?
}
```

Note: Arrow points to *next* instruction.

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

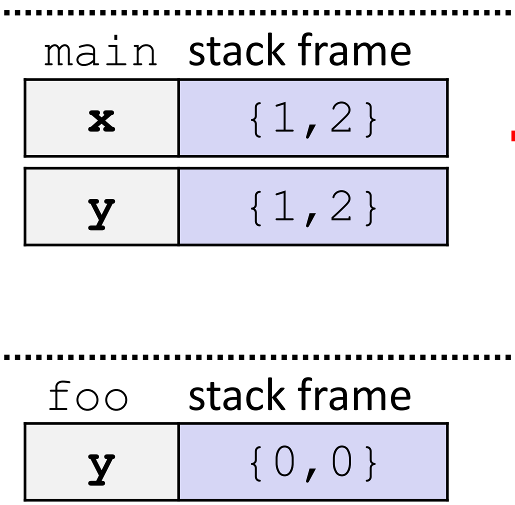



```
Point foo() {
  Point y;            // default ctor
  return y;           // copy ctor? optimized?
}

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}
y	{1,2}

`foo` stack frame

y	{0,0}

?? Temp object ??

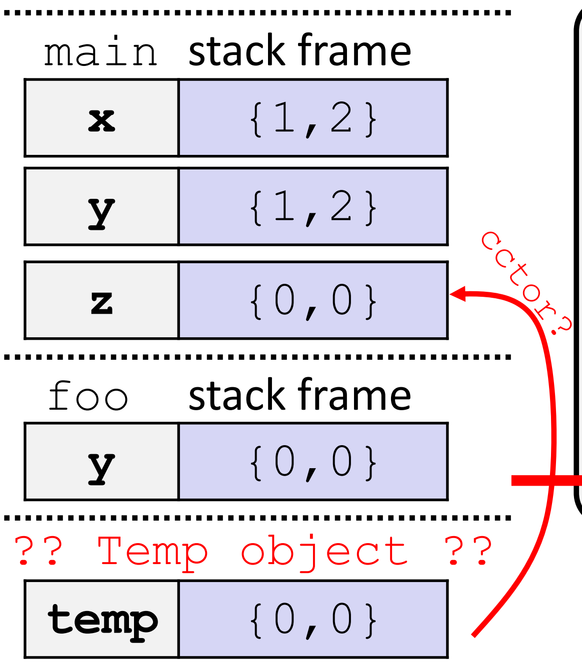
temp	{0,0}

```
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   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**
- Casting

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.

main

pt	{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;
}

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

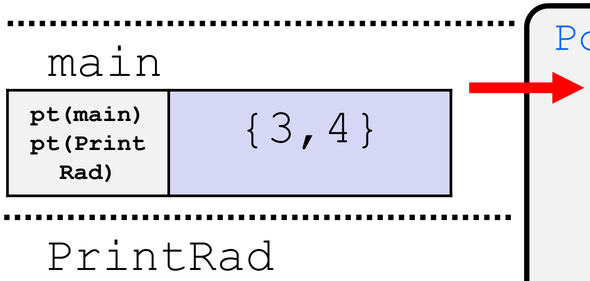
ctor	cctor	Op=	dtor
1	0	0	0

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;
}
```

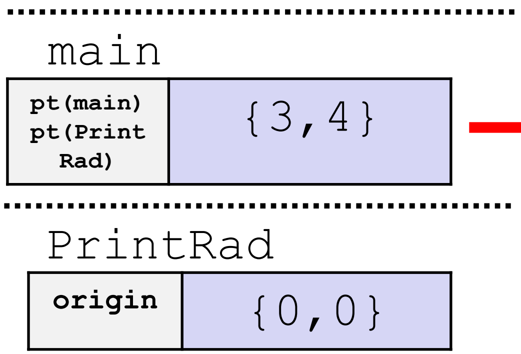
ctor	cctor	Op=	dtor
1	0	0	0

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.

main

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

PrintRad

origin	{0,0}

Point::Distance

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

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

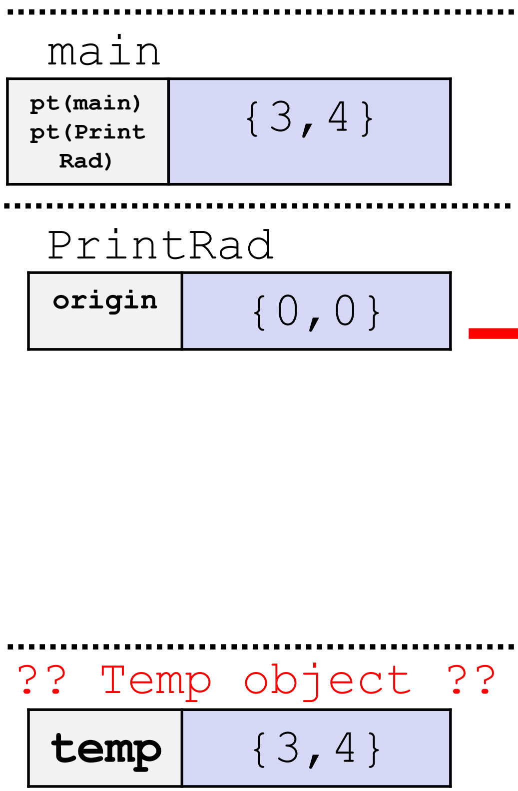
Poll Everywhere

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;
}
```

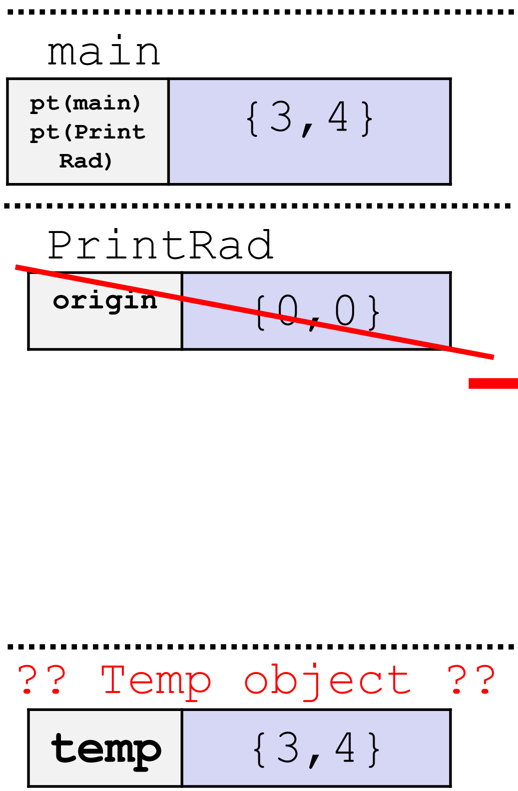
ctor	cctor	Op=	dtor
2	1	0	0

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

<u>Note</u>: 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

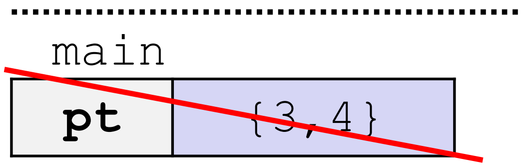
Poll Everywhere

pollev.com/tqm

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

<u>Note</u>: 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
- **Casting**

Explicit Casting in C

- Simple syntax: `lhs = (new_type) rhs;`
- Used to:
  - Convert between pointers of arbitrary type (void*) my_ptr
    - Doesn't change the data, but treats it differently
  - Forcibly convert a primitive type to another (double) my_int
    - Actually changes the representation

- You *can* still use C-style casting in C++, but sometimes the intent is not clear

Casting in C++

- C++ provides an alternative casting style that is more informative:
  - `static_cast<to_type>(expression)`
  - `dynamic_cast<to_type>(expression)`
  - `const_cast<to_type>(expression)`
  - `reinterpret_cast<to_type>(expression)`

- Always use these in C++ code
  - Intent is clearer
  - Easier to find in code via searching

static_cast

staticcast.cc

- static_cast can convert: *Any well-defined conversion*
  - Pointers to classes **of related type**
    - Compiler error if classes are not related
    - Dangerous to cast *down* a class hierarchy
  - casting `void*` to `T*`
  - Non-pointer conversion
    - *e.g.* `float` to `int`
- static_cast is checked at <u>compile time</u>

```
class A {
 public:
  int x;
};

class B {
 public:
  float y;
};

class C : public B {
 public:
  char z;
};
```

A

B → C

```
void foo() {
  B b; C c;

  // compiler error
  A* aptr = static_cast<A*>(&b);
  // OK
  B* bptr = static_cast<B*>(&c);
  // compiles, but dangerous
  C* cptr = static_cast<C*>(&b);
}
```

Unrelated types

Would have worked without cast

What happens when you do cptr->z?

dynamiccast.cc

dynamic_cast

- dynamic_cast can convert:
  - Pointers to classes **of related type**
  - References to classes **of related type**
- dynamic_cast is checked at both compile time and run time
  - Casts between unrelated classes fail at compile time
  - Casts from base to derived fail at run time if the pointed-to object is not the derived type
- Can be used like instanceof from java

```
class Base {
 public:
  virtual void foo() { }
  float x;
};

class Der1 : public Base {
 public:
  char x;
};
```

```
void bar() {
  Base b; Der1 d;

  // OK (run-time check passes)
  Base* bptr = dynamic_cast<Base*>(&d);
  assert(bptr != nullptr);

  // OK (run-time check passes)
  Der1* dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);

  // Run-time check fails, returns nullptr
  bptr = &b;
  dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);
}
```

const_cast

- `const_cast` adds or strips const-ness
  - Dangerous (!)

```
void foo(int* x) {
  *x++;
}

void bar(const int* x) {
  foo(x);                          // compiler error
  foo(const_cast<int*>(x));   // succeeds
}

int main(int argc, char** argv) {
  int x = 7;
  bar(&x);
  return EXIT_SUCCESS;
}
```

`reinterpret_cast`

- `reinterpret_cast` casts between *incompatible* types
  - Low-level reinterpretation of the bit pattern
  - *e.g.* storing a pointer in an `int`, or vice-versa
    - Works as long as the integral type is "wide" enough
  - Converting between incompatible pointers
    - Dangerous (**!**)
  - Use any other C++ cast if you can.

  - You may find it useful in HW3 (which is posted today)