References, RAII, More C++

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

pollev.com/tqm

- How are you?

Administrivia

- First Assignment (HW00 simple_string)
  - Was “Due” Friday 01/24
  - Extended to be due Wednesday the 28th (course selection period ends)
  - Mostly a C refresher
- Check-in 00
  - Short unlimited attempt quiz
  - Extended to be due Wednesday the 28th (course selection period ends)
- Check-in 01
  - Will be posted on Thursday or Friday

Administrivia

- Second Assignment (HW01 Vector)
  - Released! Should have everything you need
  - Due Friday 01/31
  - Implementing a simple C++ object

- Pre semester Survey
  - Anonymous
  - Due Wednesday the 28th

Lecture Outline

- **Error Reporting: Optional & Exceptions**
- Vector
- References
- C++ Classes
  - RAII
  - Copy Constructors
  - Copying

C++ Documentation

- As said, there is a LOT to C++
  - There are a lot of functions, objects, features, etc
  - We will NOT have time to talk about them all

- We highly recommend you make use of a C++ reference
  - cplusplus.com
    - Most find this one easier to read
    - Probably has the information you need
  - cppreference.com
    - Much more detailed (in Travis' opinion)
    - Is in various languages (scroll to the bottom)

Functions that sometimes fail

- It is pretty common to write functions that sometimes fail. Sometimes they don't return what is expected

- Consider we were building up a Queue data structure that held strings, that could
  - Add elements to the end of a sequence
    - ```
      void add(string data);
      ```
  - Remove elements from the beginning of a sequence

    ```
    ???? remove(????);
    ```
  - How do we design this function to handle the case where there are no strings in the queue (e.g. it errors?)
    ```

Previous ways to handle failing functions

- Return an "invalid" value: e.g. if looking for an index, return -1 if it can't be found.
  - What if there is no nice "invalid" state?

```
// what is an invalid string?
string remove();
```

- C-style: return an error code or success/failure.
  Real output returned through output param

```
bool remove(string* output);
```

Aside: Java "Object" variables

- Does this java compile?

```
public static String foo() {
  return null;
}
```

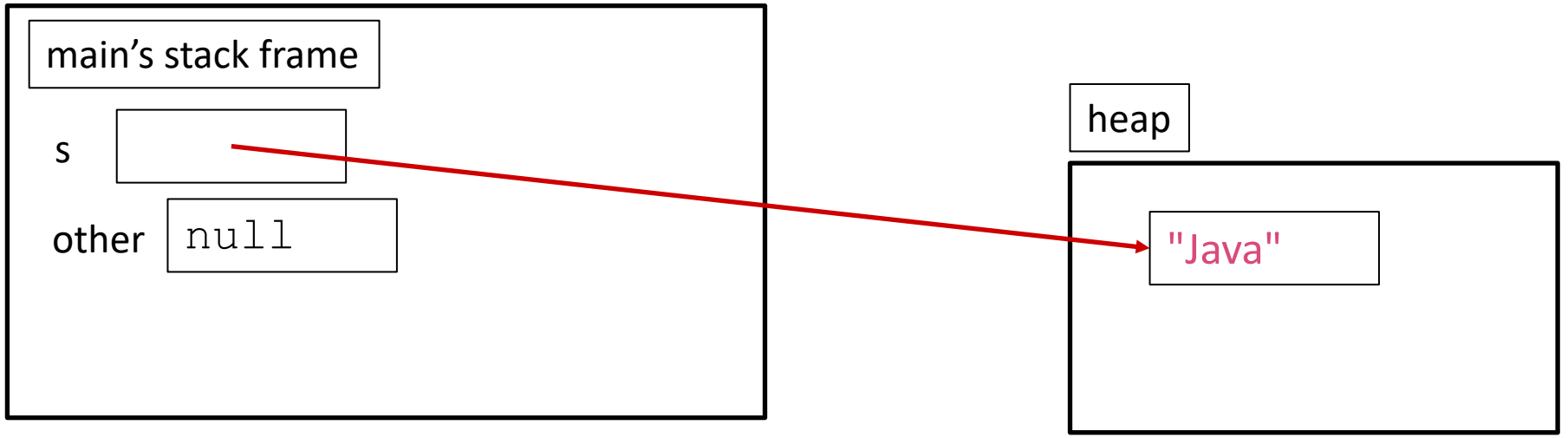
- What about this C++?

```
string foo() {
  return nullptr;
}
```

Aside: Java "Object" variables

- In high level languages (like java), object variables don't actually contain an object, they contain a reference to an object.
  - References in these languages can be null

```
String s = new String("Java");
String other = null;
```

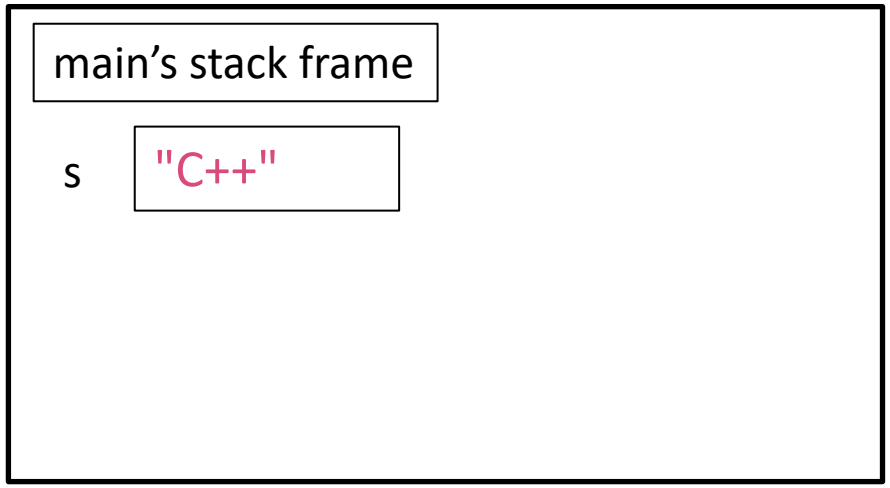


Aside: Java "Object" variables

- In C++, a string variable is itself a string object

```
string s{"C++"};

// does not do what you think it does
string other = nullptr;
```



More on this idea when I talk about pointers later

Previous ways to handle failing functions

- Return a pointer to a heap allocated object, could return **`nullptr`** on error
  - Uses the heap when it is otherwise unnecessary ☹
  - Need to remember to **`delete`** the string

```
string* remove();
```

```
string* remove() {
  if (this->size() <= 0U) {
    return nullptr;
  }
  …
  return new string(…);
```

Previous ways to handle failing functions

- "More Modern" Style: throw an exception in the case of an error return the value as normal
  - Exceptions can make your code a bit slower
  - Exception catching not always the easiest to handle
  - Exceptions aren't always the best for readability:
    - What exception(s) does this function throw?
    - Need to read the comment usually, easier to mess up :'(

```
string remove();
```

```
string remove() {
  if (this->size() <= 0U) {
    throw out_of_range("Error!");
  }
```

```
string result;
try {
  result = q.remove();
} catch (exception err) {
  // handle error
}
```

std::optional

- **`optional<T>`** is a struct that can either:
  - Have some value T (**`optional<string> {"Hello!"}`**)
  - Have nothing (**`nullopt`**)

- **`optional<T>`** effectively extends the type **`T`** to have a "null" or "invalid" state

```
optional<string> foo() {
  if (/* some error */) {
    return nullopt;
  }
  return "It worked!";
}
```

Using an optional

- If we call a function that returns an optional, we need to check to see if it has a value or not

```
optional<string> foo() {
  if (/* some error */) {
    return nullopt;
  }
  return "It worked!";
}

int main() {
  auto opt = foo();
  if (!opt.has_value()) {
    return EXIT_FAILURE;
  }
  string s = opt.value();
}
```

Lecture Outline

- Error Reporting: Optional & Exceptions
- **Vector**
- References
- C++ Classes
  - RAII
  - Copy Constructors
  - Copying

vector

C++ equivalent of ArrayList

- A generic, dynamically resizable array
  - https://cplusplus.com/reference/vector/vector/
  - Elements are store in contiguous memory locations
    - Can index into it like an array
    - Random access is O(1) time
  - Adding/removing from the end is cheap (amortized constant time)
  - Inserting/deleting from the middle or start is expensive (linear time)

- Most common member function: **`push_back()`**
  - Adds an element to the end of the vector

- **Probably the most important data structure**
  - More on this later

Vector example

```
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char* argv[]) {
  vector<int> vec {6, 5, 4};
  vec.push_back(3);
  vec.push_back(2);
  vec.push_back(1);

  cout << "vec.at(0)" << endl << vec.at(0) << endl;
  cout << "vec.at(1)" << endl << vec.at(1) << endl;

  // iterates through all elements
  for (size_t i = 0U; i < vec.size(); ++i) {
     cout << vec.at(i) << endl;
  }

  return EXIT_SUCCESS;
}
```

Most containers are in a module of the same name

Constructs a vector with three initial elements

Add three integers to the vector

Print all the values in the array

Vector iterator

```
int main(int argc, char* argv[]) {
  vector<int> vec {6, 5, 4};

  vector<int>::iterator it = vec.begin();
  it = vec.insert(it, 3);
  ++it;
  it = vec.insert(it, 1);
  it = vec.end();
  it = vec.insert(it, 2);


  cout << "Iterating:" << endl;
  for (it = vec.begin(); it < vec.end(); ++it) {
     cout << *it << endl;
  }

  return EXIT_SUCCESS;
}
```

Can get an iterator to the beginning of the vector

Insert 3 // {3, 6, 5, 4}

Advances iterator to index 1

Inserts 1 to index 1 {3, 1, 6, 5, 4}

Sets iterator to the end

Same as push_back(2);

Accesses the current element of the iterator

range for loop

- Syntactic sugar similar to Java's foreach

```
for (declaration : expression) {
  statements
}
```

  - *declaration* defines the loop variable
  - *expression* is an object representing a sequence
    - Strings, and most STL containers work with this

```
string str("hello");
// prints out each character
for (char c : str) {
  cout << c << endl;
}
```

range for loop vector example

- If you need to iterate over every element in a sequence, you should use a range for loop.
  - Why? It is harder to mess it up that way

```
int main(int argc, char* argv[]) {
  vector<int> vec {6, 5, 4};
  vec.push_back(3);
  vec.push_back(2);
  vec.push_back(1);

  // iterates through all elements
  for (int element : vec) {
     cout << element << endl;
  }

  return EXIT_SUCCESS;
}
```

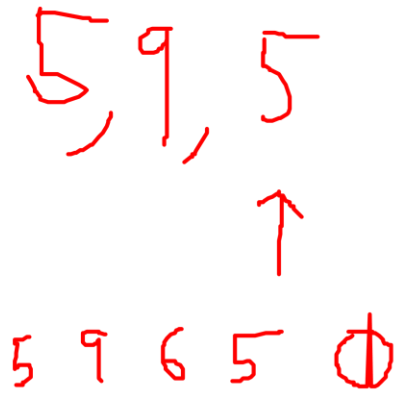
Other vector functions

- pop_back()
  - Removes the last element of the vector
- empty()
  - Returns true if the vector is empty
- clear()
  - Removes all elements currently in the vector
- erase(iterator position)
  - Erases from the element at the specified position
- A bunch more:
  - https://www.cplusplus.com/reference/vector/vector/

Poll Everywhere

pollev.com/tqm

- What is the final value of vec?




```
int main() {
  vector<int> vec {5, 9};

  vec.push_back(5);

  vector<int>::iterator it = vec.begin();
  it += 2;
  it = vec.insert(it, 6);
  it = vec.end();
  it = vec.insert(it, 0);

  for (int i : vec) {
    i *= 2;
  }
}
```

Poll Everywhere

pollev.com/tqm

- What is the final value of v by the end of the main() function?

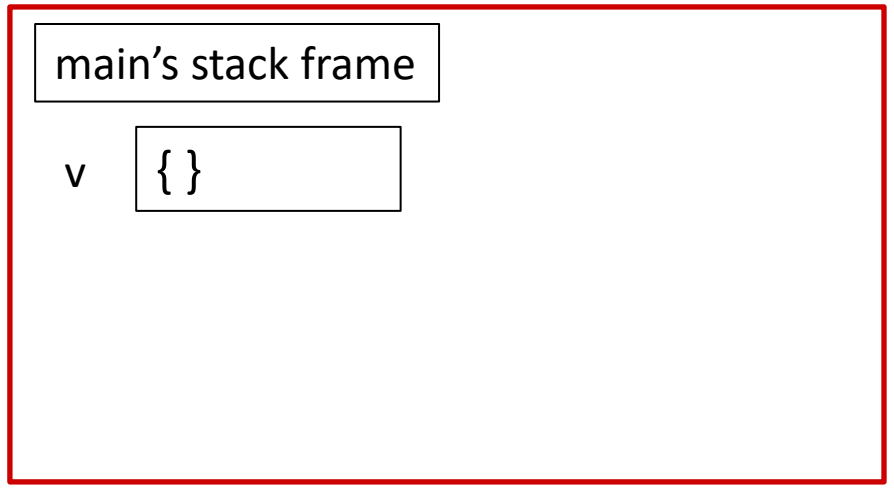
```
#include <vector>
#include <iostream>

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  for (size_t i = 0U; i < v.size(); ++i) {
     cout << v.at(i) << endl;
  }
  return EXIT_SUCCESS;
}
```

Visualization



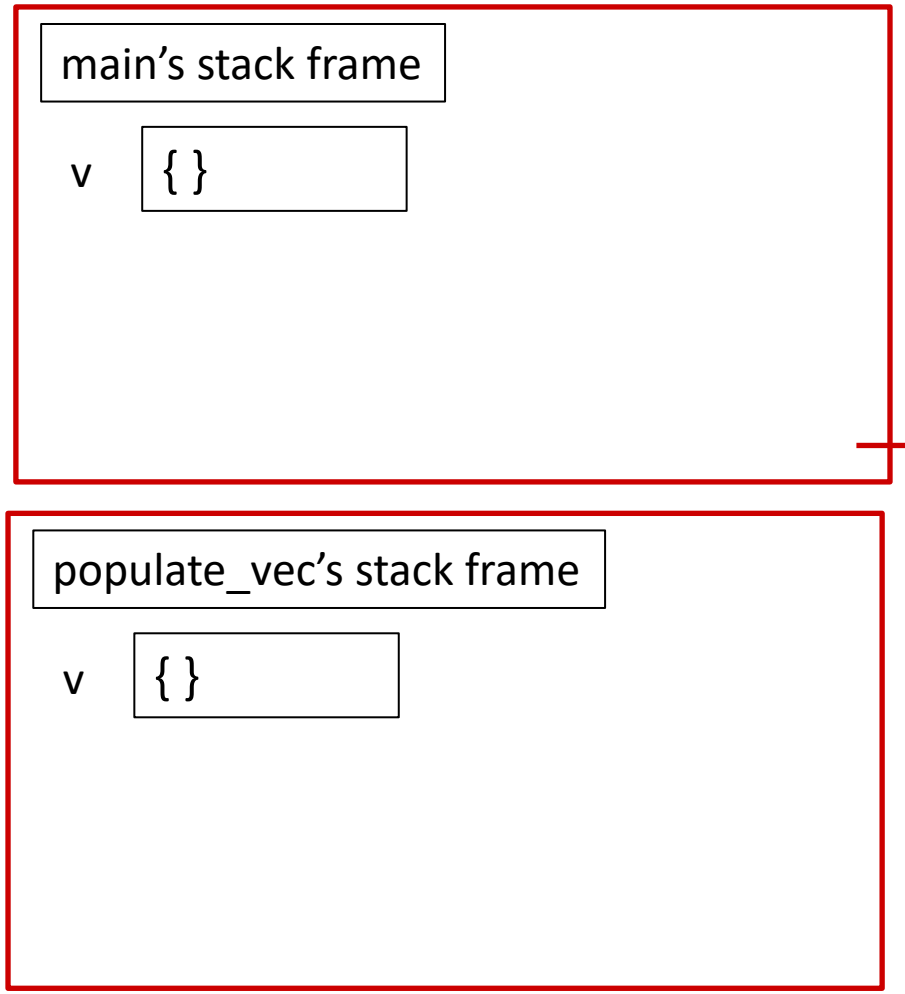

```
#include <vector>
#include <iostream>

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  // loop removed for space
  return EXIT_SUCCESS;
}
```

Visualization



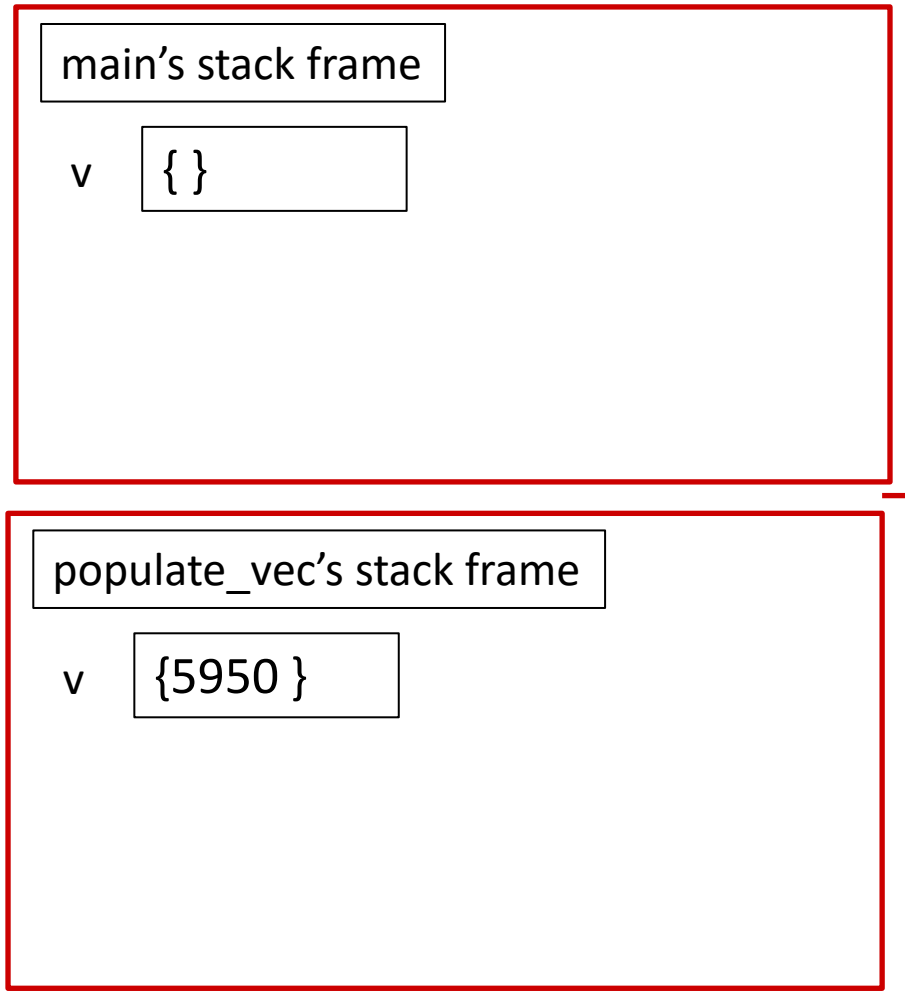

```
#include <vector>
#include <iostream>

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  // loop removed for space
  return EXIT_SUCCESS;
}
```

Visualization



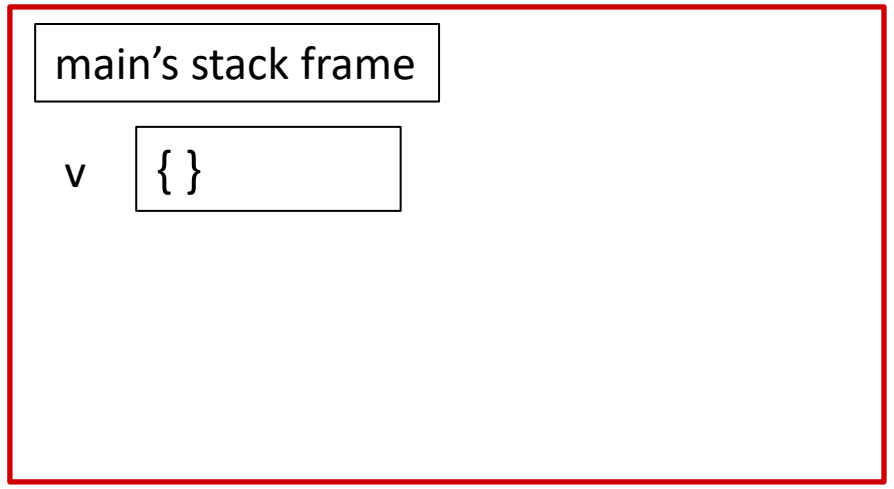

```
#include <vector>
#include <iostream>

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  // loop removed for space
  return EXIT_SUCCESS;
}
```

Visualization




```
#include <vector>
#include <iostream>

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  // loop removed for space
  return EXIT_SUCCESS;
}
```

Lecture Outline

- Error Reporting: Optional & Exceptions
- Vector
- **References**
- C++ Classes
  - RAII
  - Copy Constructors
  - Copying

References

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

- References are mostly an alternative to pointers
  - They are implemented internally with a pointer
  - But references are a lot easier to use
  - Can't do everything with references, sometimes a pointer is needed.
  - References are used a lot more often than pointers are

References

Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z  = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

When we use '&' in a type declaration, it is a reference.

x	5

y	10

References

Note: Arrow points to *next* instruction.

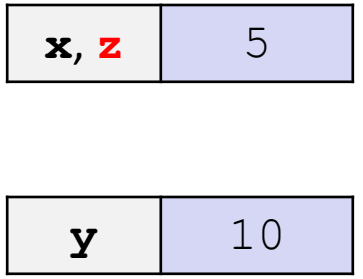
- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;
  x += 1;

  z  = y;
  z += 1;

  return EXIT_SUCCESS;
}
```



References

Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;

  z  = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

x, z	6

y	10

References

Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7

  z  = y;  // Normal assignment
  z += 1;

  return EXIT_SUCCESS;
}
```

x, z	7

y	10

References

Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7

  z  = y;  // sets z (and x) to the value of y
  z += 1;

  return EXIT_SUCCESS;
}
```

x, z	10

y	10

References

Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7

  z  = y;  // sets z (and x) to the value of y
  z += 1;  // sets z (and x) to 11

  return EXIT_SUCCESS;
}
```

x, z	11

y	10

Pass-By-Reference

Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```

Parameters are attached To variables provided by caller

(main) **a**	5

(main) **b**	10

Pass-By-Reference

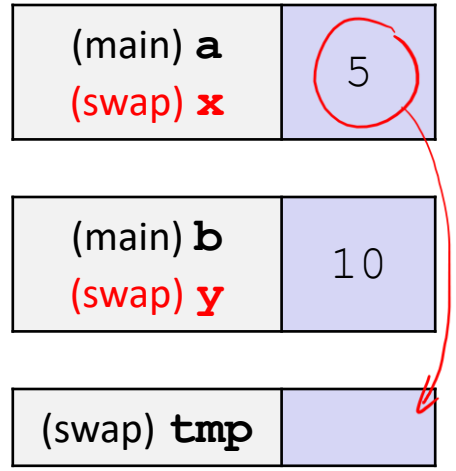
Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```



Pass-By-Reference

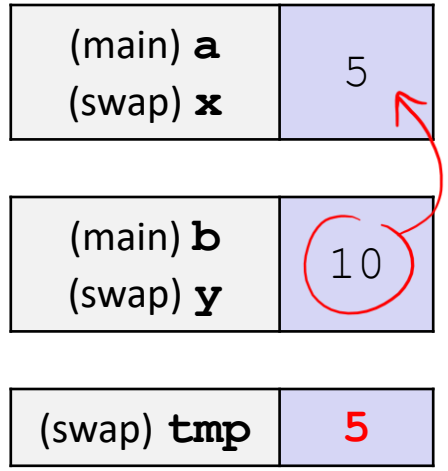
Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```



Pass-By-Reference

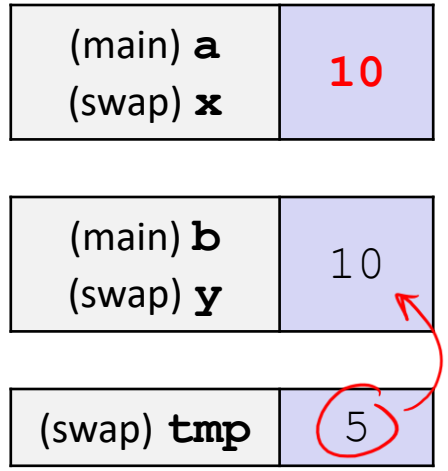
Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```



Pass-By-Reference

Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```

(main) **a** (swap) **x**	10
(main) **b** (swap) **y**	**5**
(swap) **tmp**	5

Pass-By-Reference

Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```

(main) **a**	10

(main) **b**	5

Pass-By-Reference

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

- Can use on objects as well!

- Now v is actually changed"

```
void populate_vec(vector<int>& v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  // loop removed for space
  return EXIT_SUCCESS;
}
```

Pass-By-Reference

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

- Can use in range for loops as well!

- Now vector values are actually changed"

```
int main() {
  vector<int> vec {5, 9};

  for (int& i : vec) {
    i *= 2;
  }
}
```

Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

A. Output "(3,3,3)"
B. Output "(3,3,2)"
C. Output "(3, 3, 1)"
D. Compiler error
E. We're lost…

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

Polling Question

- What will happen when we run this?

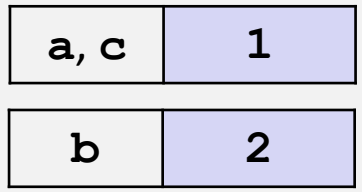
Note: Arrow points to *next* instruction.

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```



Polling Question

- What will happen when we run this?

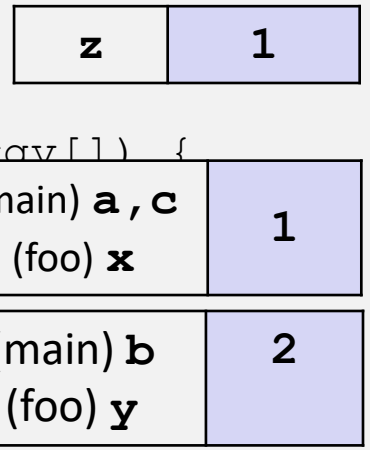
Note: Arrow points to *next* instruction.

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```



Polling Question

- What will happen when we run this?

Note: Arrow points to *next* instruction.

```
void foo(int& x, int& y, int z) {
  z = y;
→ x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

z	2

(main) a, c (foo) x	1

(main) b (foo) y	2

Polling Question

- What will happen when we run this?

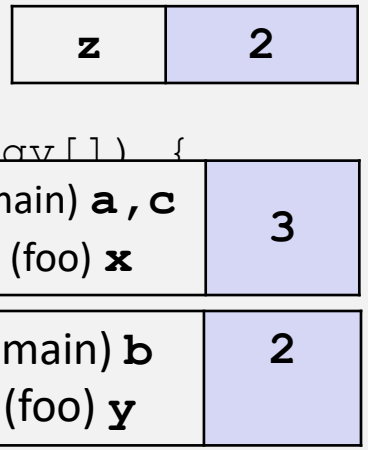
Note: Arrow points to *next* instruction.

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```



Polling Question

- What will happen when we run this?

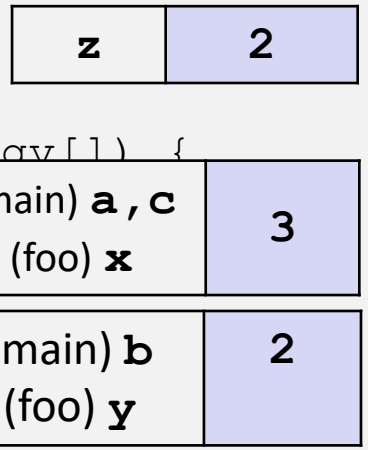
Note: Arrow points to *next* instruction.

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```



Polling Question

- What will happen when we run this?

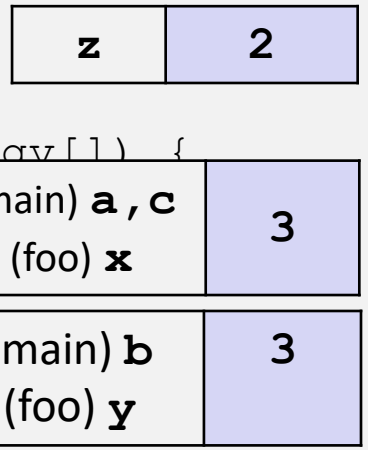
Note: Arrow points to *next* instruction.

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```



Polling Question

- What will happen when we run this?

Note: Arrow points to *next* instruction.

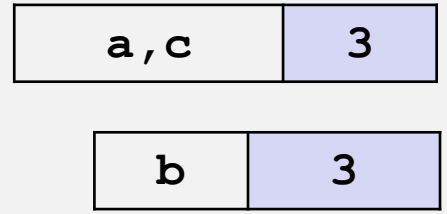
A. Output "(3,3,3)"
B. Output "(3,3,2)"
C. Output "(3, 3, 1)"
D. Compiler error
E. We're lost…

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```



Lecture Outline

- Error Reporting: Optional & Exceptions
- Vector
- References
- **C++ Classes**
  - **RAII**
  - Copy Constructors
  - Copying

short discussion

- This code uses the heap
- Does this code memory leak?

```
#include <vector>
#include <iostream>

using namespace std;

int main() {
  vector<int> v {333, 484};
  v.push_back(5950);
  cout << v.size() << endl;
  for (size_t i = 0U; i < v.size(); ++i) {
     cout << v.at(i) << endl;
  }
  return EXIT_SUCCESS;
}
```

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

More on RAII in a later lecture

```
MyObj::~MyObj() {    // destructor
  // do any cleanup needed when a "MyObj" 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 Integer {
 public:
  Integer(int val) {     // Constructor
    val_ = new int(val);
  }

  ~Integer() { delete val_; }   // Destructor
  int get_value() { return *val_; }   // inline member function
 private:
  int* val_;  // data member
};  // class Integer
```

Integer.h

Allocates memory in the constructor

Without destructor, the memory wouldn't be freed

```
#include "Integer.h"
#include <iostream>

int main(int argc, char** argv) {
  Integer best_course{5950};
  cout << best_course.get_value() << endl;
  return EXIT_SUCCESS;
}
```

Destruct the object when it falls out of scope (when we return)

Default Destructor

- 9 out of ten times, most objects do not need to create an explicit destructor.
- Destructors can be specified to be a default the C++ generates for you
- The default destructor just runs the destructor of any data member (fields) the object has.
  - So, if your custom object has a vector or a map, then those data structures will automatically get destructed/"cleaned-up"

Default Destructor Example

```
#ifndef POINT_HPP_
#define POINT_HPP_

class Point {
 public:
  Point(int x, int y);     // constructor
  ~Point() = default;
  int get_x() { return x_; }      // inline member function
  int get_y() { return y_; }      // inline member function
  double Distance(Point p);       // member function
  void SetLocation(int x, int y); // member function

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

#endif  // POINT_HPP_
```

Default destructor since we don't do any allocation in Point

Lecture Outline

- Error Reporting: Optional & Exceptions
- Vector
- References
- **C++ Classes**
  - RAII
  - **Copy Constructors**
  - **Copying**

Copy Constructors

- C++ has the notion of a copy constructor (cctor)
  - Used to create a new object as an independent 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

Synthesized Copy Constructor

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

Calls cctor of data members that are objects

Does assignment for primitives
Could be problematic with pointers

```
#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;
}
```

Synthesized Copy Constructor: Example

- If you don't define your own copy constructor, C++ will synthesize one for you
  - It will do a *shallow* copy of all of the fields (*i.e.* member variables) of your class
  - Sometimes the right thing; sometimes the wrong thing
  - This is an example of what the synthesized copy constructor would look like. Is this correct? **pollev.com/tqm**

Calls cctor of data members that are objects

Does assignment for primitives
Could be problematic with pointers

```
class Integer {
 public:
  Integer(int val) {     // Constructor
    val_ = new int(val);
  }
  Integer(const Integer& other) {
    val_ = other.val_;
  }
  ~Integer() { delete val_; }   // Destructor
  int get_value() { return *val_; }   // inline member function
 private:
  int* val_;  // data member
};  // class Integer
```

Poll Everywhere

pollev.com/tqm

- This is an example of what the synthesized copy constructor would look like. Is this correct?

pollev.com/tqm

```
class Integer {
 public:
  Integer(int val) {     // Constructor
    val_ = new int(val);
  }
  Integer(const Integer& other) {
    val_ = other.val_;
  }
  ~Integer() { delete val_; }  // Destructor
  int get_value() { return *val_; }
 private:
  int* val_;  // data member
};  // class Integer
```

```
int main() {
  Integer x(5950);
  Integer y(x);

  cout << y.get_value() << endl;
  cout << x.get_value() << endl;
}
```

Synthesized Copy Constructor: Example

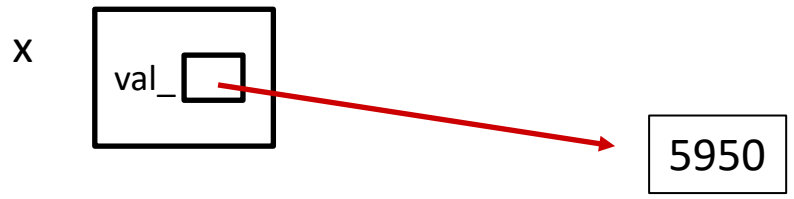
- This is an example of what the synthesized copy constructor would look like. Is this correct?

pollev.com/tqm

```
class Integer {
 public:
  Integer(int val) {     // Constructor
    val_ = new int(val);
  }
  Integer(const Integer& other) {
    val_ = other.val_;
  }
  ~Integer() { delete val_; }  // Destructor
  int get_value() { return *val_; }
 private:
  int* val_;  // data member
};  // class Integer
```

```
int main() {
  Integer x(5950);
  Integer y(x);

  cout << y.get_value() << endl;
  cout << x.get_value() << endl;
}
```



Synthesized Copy Constructor: Example

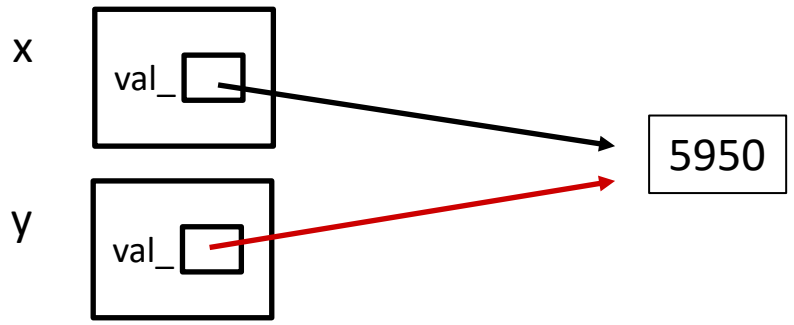
- This is an example of what the synthesized copy constructor would look like. Is this correct?

pollev.com/tqm

```
class Integer {
 public:
  Integer(int val) {     // Constructor
    val_ = new int(val);
  }
  Integer(const Integer& other) {
    val_ = other.val_;
  }
  ~Integer() { delete val_; }  // Destructor
  int get_value() { return *val_; }
 private:
  int* val_;  // data member
};  // class Integer
```

```
int main() {
  Integer x(5950);
  Integer y(x);

  cout << y.get_value() << endl;
  cout << x.get_value() << endl;
}
```



Synthesized Copy Constructor: Example

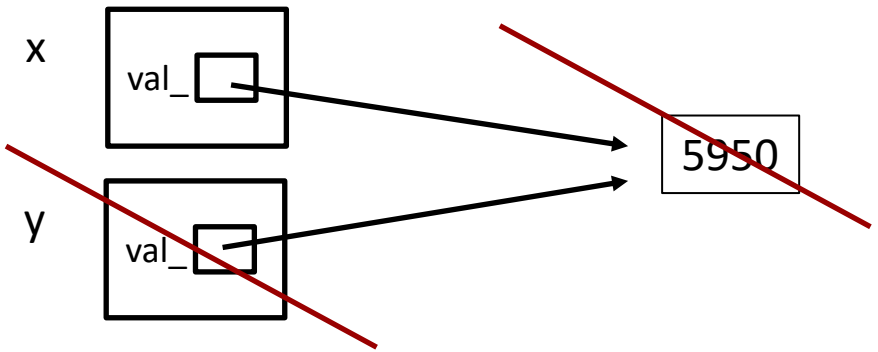
- This is an example of what the synthesized copy constructor would look like. Is this correct?

pollev.com/tqm

```
class Integer {
 public:
  Integer(int val) {     // Constructor
    val_ = new int(val);
  }
  Integer(const Integer& other) {
    val_ = other.val_;
  }
  ~Integer() { delete val_; }  // Destructor
  int get_value() { return *val_; }
 private:
  int* val_;  // data member
};  // class Integer
```

```
int main() {
  Integer x(5950);
  Integer y(x);

  cout << y.get_value() << endl;
  cout << x.get_value() << endl;
}
```



Synthesized Copy Constructor: Example

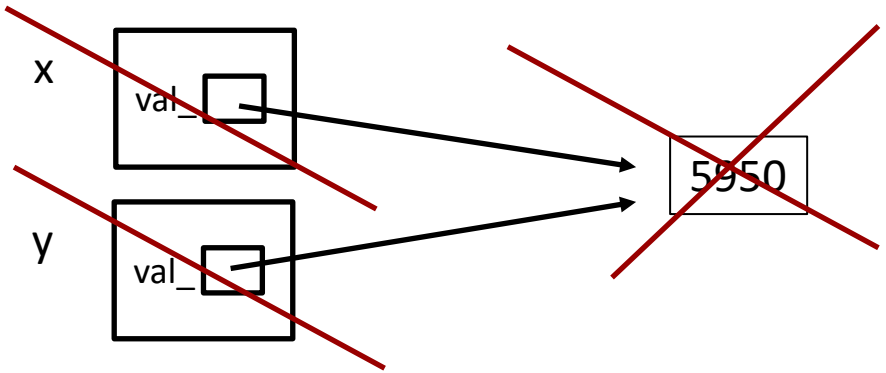
- This is an example of what the synthesized copy constructor would look like. Is this correct?

pollev.com/tqm

```
class Integer {
 public:
  Integer(int val) {     // Constructor
    val_ = new int(val);
  }
  Integer(const Integer& other) {
    val_ = other.val_;
  }
  ~Integer() { delete val_; }  // Destructor
  int get_value() { return *val_; }
 private:
  int* val_;  // data member
};  // class Integer
```

```
int main() {
  Integer x(5950);
  Integer y(x);

  cout << y.get_value() << endl;
  cout << x.get_value() << endl;
}
```



They are not independent, and we get a double delete error

Copy Constructor: Fixed

- Fixed Copy Constructor :)

```
class Integer {
 public:
  Integer(int val) {     // Constructor
    val_ = new int(val);
  }
  Integer(const Integer& other) {
    val_ = new int(other.get_value());
  }
  ~Integer() { delete val_; }  // Destructor
  int get_value() { return *val_; }
 private:
  int* val_;  // data member
};  // class Integer
```

Poll Everywhere

pollev.com/tqm

- What does this code print?

```
int main() {
  vector<int> v {3, 5};
  v.push_back(2);

  vector<int> plato = v;
  plato.push_back(16);

  for (int i : v) {
    cout << i << endl;
  }
}
```

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
}
```

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

```
class Point {
 public:
  // ctor
  Point(int x, int y);
  // cctor
  Point(const Point& other);

  // doesn't make any copies
  Distance(const Point& other);
};
```

```
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);
  Point p = PrintRad(pt);
  return 0;
}
```

A. 1
B. 2
C. 3
D. 4
E. We're lost…

Polling Question

- 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);
  Point p = PrintRad(pt);
  return 0;
}
```

ctor	cctor	dtor
0	0	0

Polling Question

- 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}

```
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;
}

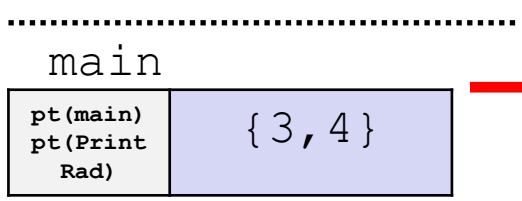
int main(int argc, char** argv) {
  Point pt(3, 4);
  Point p = PrintRad(pt);
  return 0;
}
```

ctor	cctor	dtor
1	0	0

Polling Question

- 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.




```
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;
}

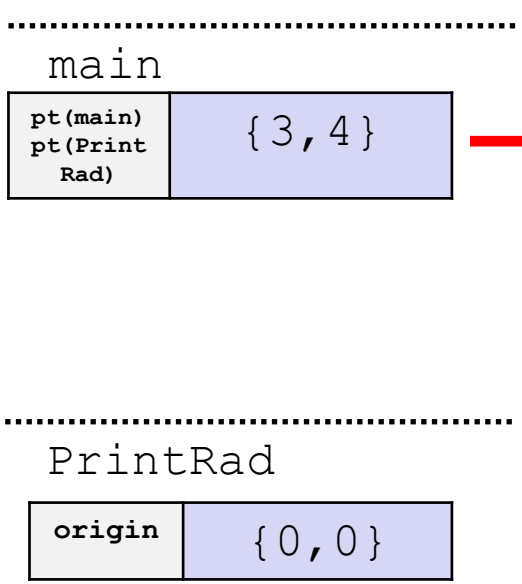
int main(int argc, char** argv) {
  Point pt(3, 4);
  Point p = PrintRad(pt);
  return 0;
}
```

ctor	cctor	dtor
1	0	0

Polling Question

- 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.




```
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;
}

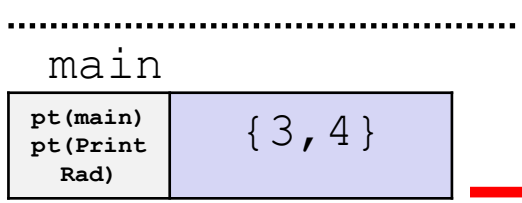
int main(int argc, char** argv) {
  Point pt(3, 4);
  Point p = PrintRad(pt);
  return 0;
}
```

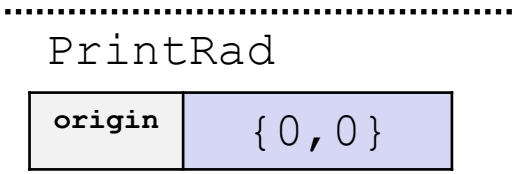
ctor	cctor	dtor
2	0	0

Polling Question

- 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.





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;
}

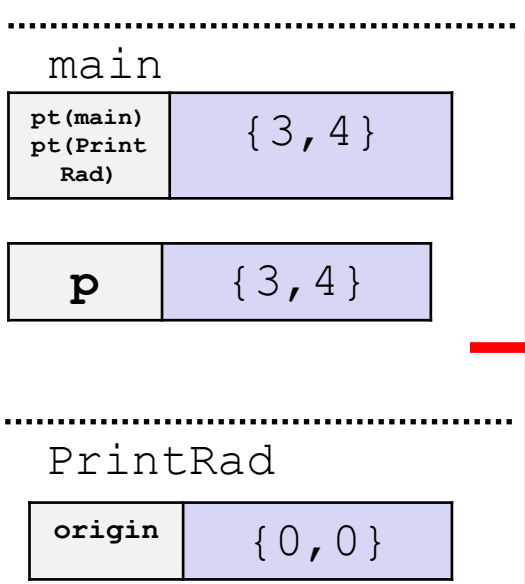
int main(int argc, char** argv) {
  Point pt(3, 4);
  Point p = PrintRad(pt);
  return 0;
}
```

ctor	cctor	dtor
2	0	0

Polling Question

- 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.




```
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;
}

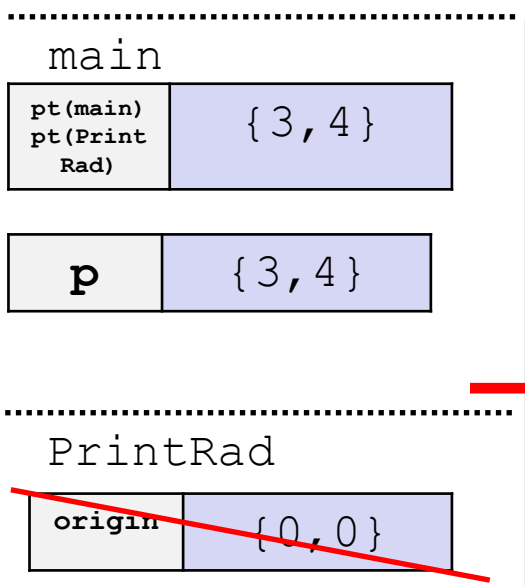
int main(int argc, char** argv) {
  Point pt(3, 4);
  Point p = PrintRad(pt);
  return 0;
}
```

ctor	cctor	dtor
2	1	0

Polling Question

- 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.




```
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;
}

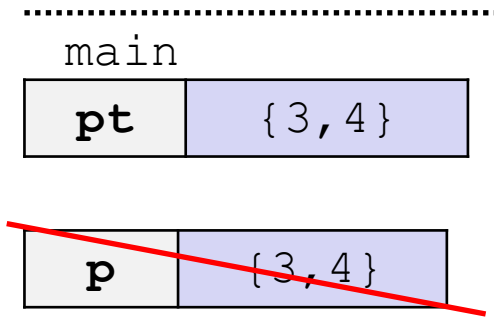
int main(int argc, char** argv) {
  Point pt(3, 4);
  Point p = PrintRad(pt);
  return 0;
}
```

ctor	cctor	dtor
2	1	1

Polling Question

- 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.




```
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;
}

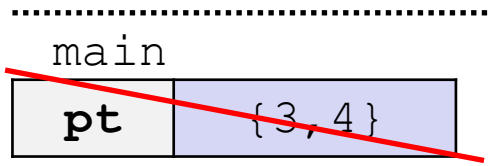
int main(int argc, char** argv) {
  Point pt(3, 4);
  Point p = PrintRad(pt);
  return 0;
}
```

ctor	cctor	dtor
2	1	2

Polling Question

- 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.




```
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);
  Point p = PrintRad(pt);
  return 0;
}
```

C. 3

ctor	cctor	dtor
2	1	3

That's it for now!

- More next lecture ☺
  - Copying
  - Copying Overhead
  - Lifetimes
  - std::variant
  - And more!