# C++ Classes & References

## Computer Systems Programming, Spring 2023


**Instructor:** Travis McGaha

**TAs:**

| | |
|---|---|
| Kevine Bernat | Jialin Cai |
| Mati Davis | Donglun He |
| Chandravaran Kunjeti | Heyi Liu |
| Shufan Liu | Eddy Yang |

# Logistics

- HW00: **Due TOMORROW 1/26 @ 11:59 PM**
  - Implement LinkedList & HashTable
  - You should have everything you need
  - HWs can take a while
  - **DO NOT PUT THIS OFF, PLEASE GET STARTED**

- HW1 (FileReaders)
  - To be released shortly after HW1

# Poll Everywhere

pollev.com/tqm

- What happens if someone tried to compile and run the following code?

```
#include <iostream>

void foo(const int* ptr) {
  int temp = *ptr + 20;
  std::cout << temp << std::endl;
}

int main() {
  const int x = 5930;
  foo(&x);
}
```

A. Compiles, but gets Segmentation fault

B. Compilation Error

C. Prints "5930"

D. Prints "5950"

E. We're lost…

# Poll Everywhere

pollev.com/tqm

- Does the following code compile?

```
#include <iostream>

int main() {
  std::cout << "Hello CIT" << ' ' << 5950 << "!!!" << std::endl;
}
```

A. Yes

B. No

C. We're lost…

# Poll Everywhere

pollev.com/tqm

- Does the following code compile?

A. Yes
B. No
C. We're lost...

```
#include <iostream>

using std::cout;

int main() {
  cout << "Hello CIT" << ' ';
  cout << 5950 << "!!!" << endl;
}
```

# Lecture Outline

- **C++ Objects**
- Constructors & Destructors
- References in C++

# Structs in C

- In C, we only had **`struct`**s, which could only bundle together data fields

- Struct example definition:

```
struct Point {  // Declare struct, usually used typedef
  // Declare fields & types here
  int x;
  int y;
};
```

- What is missing from this compared to objects/classes in languages other languages?
  - Methods
  - Access modifiers (public vs private)
  - Inheritance

# Classes in C++

- In C++, we have classes.
  - Think of these as C structs, but with methods, access modifiers, and inheritance.
- Class example definition:

Similar syntax for declaration

```
class Point {  // Declare class, typedef usually not used
 public:
  Point(int x, int y);  // constructor
  int get_x();          // getter
  int get_y();          // getter
 private:
  int x_;               // fields
  int y_;
};
```

Access modifiers

methods

Fields

- In C++, we call fields and methods "members"

# Classes Syntax

- Class definition syntax (in a `.h` file):

```
class Name {
 public:
  // public member definitions & declarations go here

 private:
  // private member definitions & declarations go here
};  // class Name
```

don't forget!

  - Members can be functions (methods) or data (variables)

- Class member function definition syntax (in a `.cc` file):

```
retType Name::MethodName(type1 param1, …, typeN paramN) {
  // body statements
}
```

  - (1) *define* within the class definition or (2) *declare* within the class definition and then *define* elsewhere

# Class Definition (.h file)

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(int x, int y);      // constructor
  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_H_
```

Declarations

Inline definition ok for simple getters/setters

C++ naming conventions for data members

# Class Member Definitions (.cc file)

Point.cc

This code uses bad style for demonstration purposes

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

Point::Point(int x, int y) {
  x_ = x;
  this->y_ = y;  // "this->" is optional unless name conflicts
}

double Point::Distance(Point p) {
  // 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);
}

void Point::SetLocation(int x, int y) {
  x_ = x;
  y_ = y;
}
```

Equivalent to y_=y;

"this" is a Point* const

We have access to x_, could have used x_ instead.

# 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)

# Lecture Outline

- C++ Objects
- **Constructors & Destructors**
- References in C++

# 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);
```

  - C++ will automatically create a synthesized default constructor if you have ***no*** user-defined constructors
    - Takes no arguments and calls the default ctor on all non-"plain old data" (non-POD) member variables
    - Synthesized default ctor will fail if you have non-initialized const or reference data members

Zero arg

Created for you

# Synthesized Default Constructor Example

SimplePoint.h

```
class SimplePoint {
 public:
  // no constructors declared!
  int get_x() { return x_; }      // inline member function
  int get_y() { return y_; }      // inline member function
  double Distance(SimplePoint p);
  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 (zero arg) ctor

SimplePoint.cc

```
#include "SimplePoint.h"

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

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

# 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(int x, int y) {
  x_ = x;
  y_ = y;
}

void foo() {
  SimplePoint x;        // compiler error:  if you define any
                        // ctors, C++ will NOT synthesize a
                        // default constructor for you.

  SimplePoint y(1, 2);  // works:  invokes the 2-int-arguments
                        // constructor
}
```

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(int x, 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(int x, int y) {
  x_ = x;
  y_ = y;
  std::cout << "Point constructed: (" << x_ << ",";
  std::cout << y_<< ")" << std::endl;
}
```

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

# Initialization vs. Construction

```
class Point3D {
 public:
  // constructor with 3 int arguments
  Point3D(int x, int y, int z) : y_(y), x_(x) {
    z_ = z;
  }

 private:
  int x_, y_, z_;  // data members
};  // class Point3D
```

*First*, initialization list is applied.

2) set y_

1) set x_

3) set z_ (garbage)

4) set z_

*Next*, constructor body is executed.

- 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

# 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 Integer {
 public:
  Integer(int val) : val_(new int(val)) {       // Constructor
  }

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

Without destructor, the memory wouldn't be freed

Integer.h

```
#include "Integer.h"


int main(int argc, char** argv) {
  Integer best_course(595);
  return EXIT_SUCCESS;
}
```

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

# Lecture Outline

- C++ Objects
- Constructors & Destructors
- **References in C++**

# Poll Everywhere

pollev.com/tqm

- What is printed in this example?

A. Output "5930"

B. Output "5950"

C. Segmentation fault

D. Compiler error about body of foo

E. We're lost…

```
#include <iostream>
#include <cstdlib>

void foo(int x) {
  x = 5950;
}

int main(int argc, char** argv) {
  int x = 5930;
  foo(x);
  std::cout << x << std::endl;
}
```

# Poll Everywhere

pollev.com/tqm

- What is printed in this example?

A. Output "5930"
B. Output "5950"
C. Segmentation fault
D. Compiler error about body of foo
E. We're lost…

```
#include <iostream>
#include <cstdlib>

void foo(int x) {
  x = 5950;
}

int main(int argc, char** argv) {
  int x = 5930;
  foo(x);
  std::cout << x << std::endl;
}
```

# Poll Everywhere

pollev.com/tqm

- What is printed in this example?

A. Output "5930"

B. Output "5950"

C. Segmentation fault

D. Compiler error about body of foo

E. We're lost…

```
#include <iostream>
#include <cstdlib>

class Integer {
 public:
  Integer(int val) : val_(val) {}
  int get_value() { return val_; }
  void set_value(int val) { val_ = val;}
 private:
  int val_;
};

void foo(Integer x) { x.set_value(5950); }

int main(int argc, char** argv) {
  Integer x(5930);
  foo(x);
  std::cout << x.get_value() << std::endl;
  return EXIT_SUCCESS;
}
```

# Poll Everywhere

**pollev.com/tqm**

- What is printed in this example?

A. Output "5930"
B. Output "5950"
C. Segmentation fault
D. Compiler error about body of foo
E. We're lost...

```
#include <iostream>
#include <cstdlib>

class Integer {
 public:
  Integer(int val) : val_(val) {}
  int get_value() { return val_; }
  void set_value(int val) { val_ = val;}
 private:
  int val_;
};

void foo(Integer x) { x.set_value(5950); }

int main(int argc, char** argv) {
  Integer x(5930);
  foo(x);
  std::cout << x.get_value() << std::endl;
  return EXIT_SUCCESS;
}
```

# Pass-by-value

- C++ is pass-by-value on default
  - **This includes objects**

- When an object is passed as a parameter normally, a copy is created and passed into the function instead.
  - The semantics of copying an object can be complicated, will be discussed later in the course

- There is a way to enable pass-by-reference...

# Pointers Reminder

Note: Arrow points to *next* instruction.

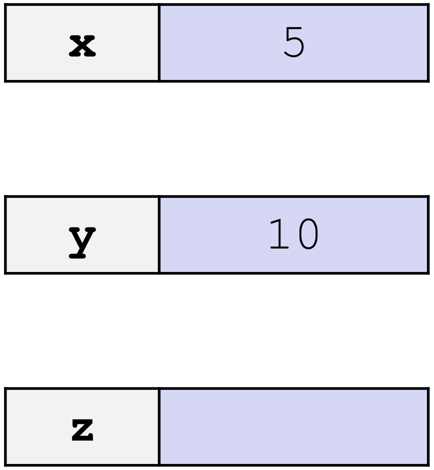
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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;
}
```



# Pointers Reminder

Note: Arrow points to *next* instruction.

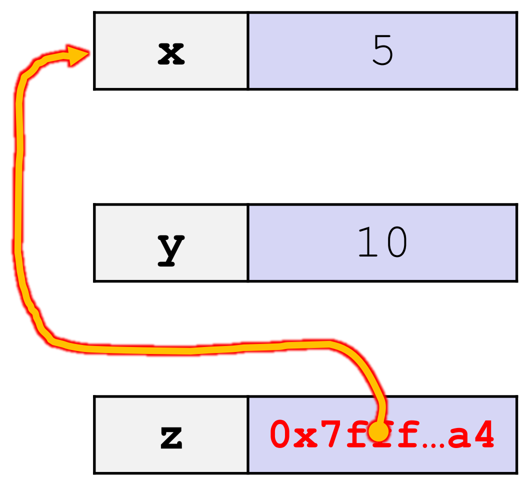
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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;
}
```



# Pointers Reminder

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

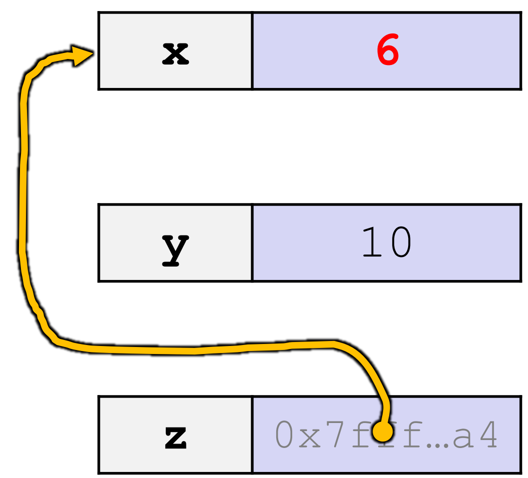
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

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

  *z += 1;  // sets x to 6
   x += 1;

   z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



# Pointers Reminder

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

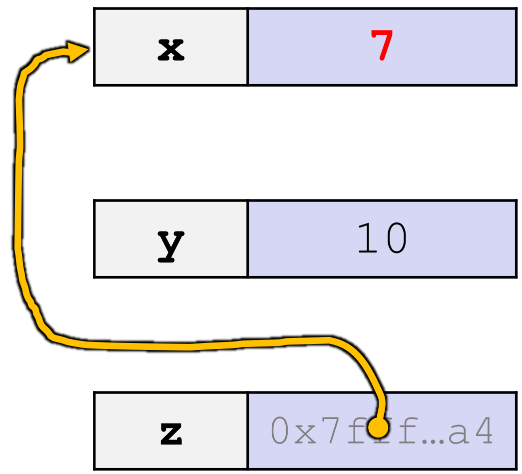
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

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

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

   z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



# Pointers Reminder

Note: Arrow points to *next* instruction.

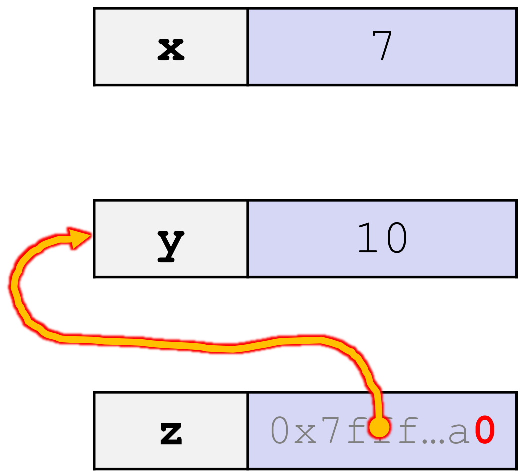
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

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

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

   z = &y;   // sets z to the address of y
  *z += 1;

  return EXIT_SUCCESS;
}
```



# Pointers Reminder

Note: Arrow points to *next* instruction.

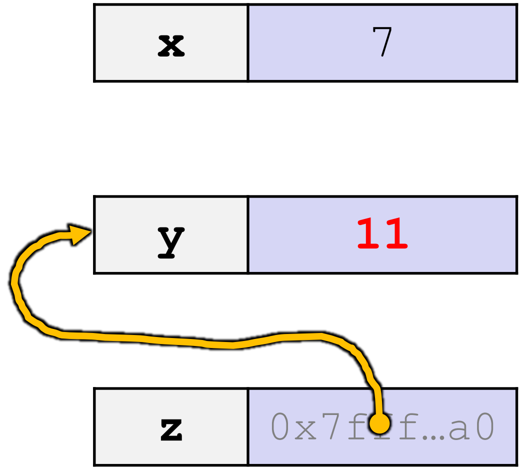
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

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

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

   z = &y;  // sets z to the address of y
  *z += 1;  // sets y (and *z) to 11

  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;

  z += 1;
  x += 1;

  z  = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

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

&var still is "address of var"

| | |
|---|---|
| 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;
}
```

| | |
|---|---|
| **x, z** | 5 |
| **y** | 10 |

# References

Note: Arrow points to *next* instruction.

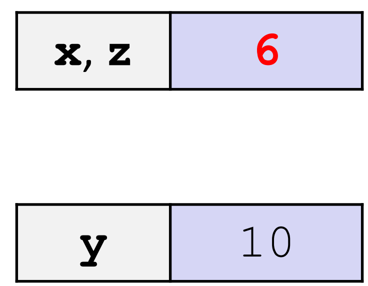
- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

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

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

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

  return EXIT_SUCCESS;
}
```

| | |
|---|---|
| **x, z** | 7 |
| **y** | 10 |

# References

<u>Note</u>: 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

<u>Note</u>: 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.

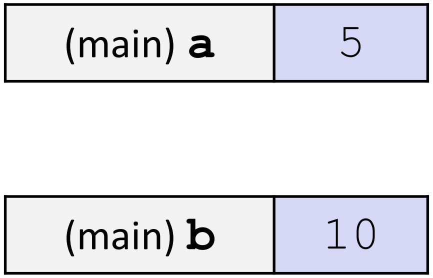
- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
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



# Pass-By-Reference

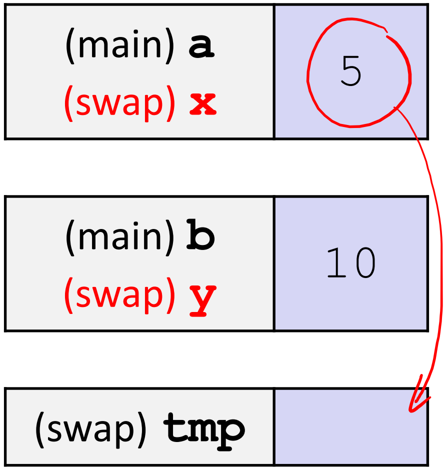
Note: Arrow points to *next* instruction.

- 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;
}
```



Note: Arrow points to *next* instruction.

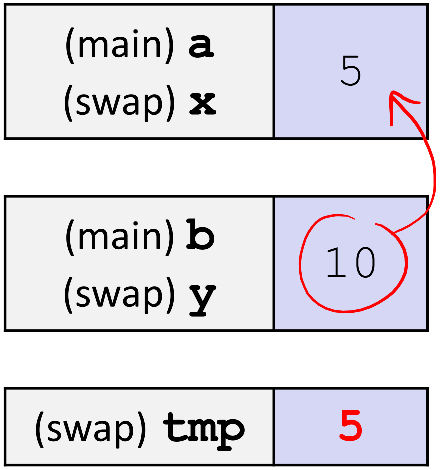
# Pass-By-Reference

- 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!

```
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;
}
```



Note: Arrow points to *next* instruction.

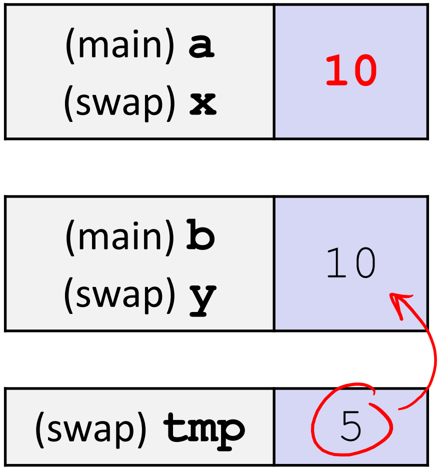
# Pass-By-Reference

- 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!

```
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

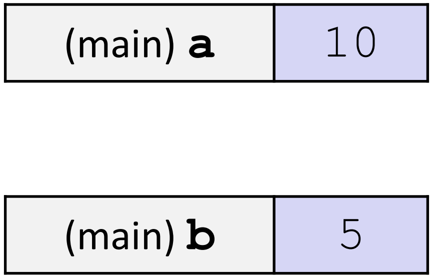
Note: Arrow points to *next* instruction.

- 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;
}
```



# 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 prints "595"

```
void foo(Integer& x) {
  x.set_value(595);
}

int main(int argc, char** argv) {
  Integer x(593);
  foo(x);
  std::cout << x.get_value() << std::endl;
  return EXIT_SUCCESS;
}
```

# Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

A. Output "(1,2,3)"
B. Output "(3,2,3)"
C. Compiler error about arguments to foo (in main)
D. Compiler error about body of foo
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);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```

# Poll Everywhere

**pollev.com/tqm**

- What will happen when we run this?

Note: Arrow points to *next* instruction.

A. Output "(1,2,3)"

B. Output "(3,2,3)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

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);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```

| | |
|---|---|
| a, c | 1 |
| b | 2 |

# Poll Everywhere

**pollev.com/tqm**

- What will happen when we run this?

Note: Arrow points to *next* instruction.

**A.** Output "(1,2,3)"

**B.** Output "(3,2,3)"

**C.** Compiler error about arguments to foo (in main)

**D.** Compiler error about body of foo

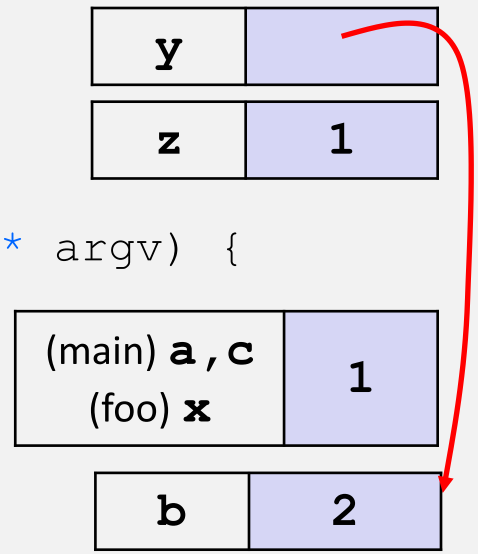
**E.** We're lost…

```
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);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```



# Poll Everywhere

**pollev.com/tqm**

- What will happen when we run this?

Note: Arrow points to *next* instruction.

A. Output "(1,2,3)"

B. Output "(3,2,3)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

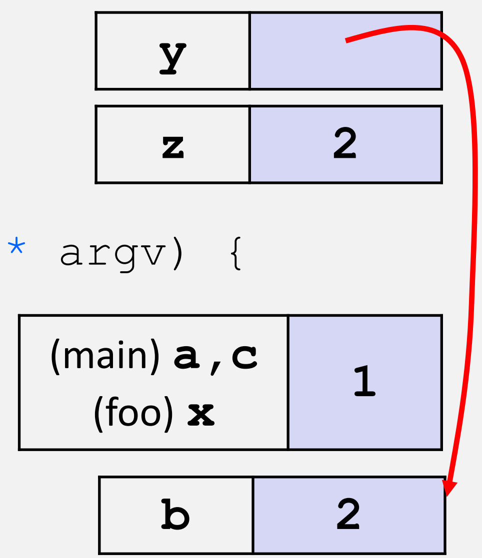
E. We're lost…

```
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);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```



# Poll Everywhere

**pollev.com/tqm**

- What will happen when we run this?

Note: Arrow points to *next* instruction.

A. Output "(1,2,3)"

B. Output "(3,2,3)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

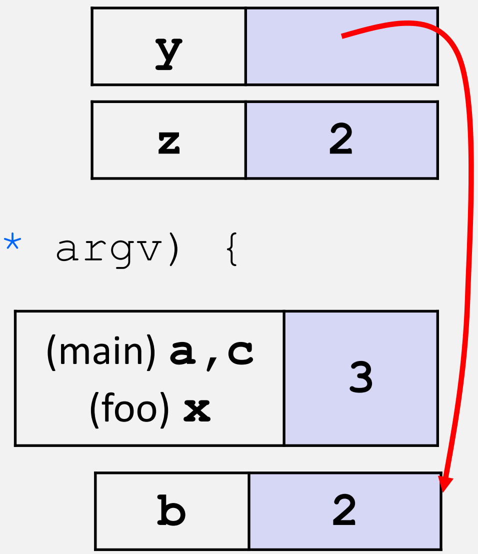
E. We're lost…

```
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);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```



# Poll Everywhere

**pollev.com/tqm**

- What will happen when we run this?

Note: Arrow points to *next* instruction.

**A. Output "(1,2,3)"**

**B. Output "(3,2,3)"**

**C. Compiler error about arguments to foo (in main)**

**D. Compiler error about body of foo**

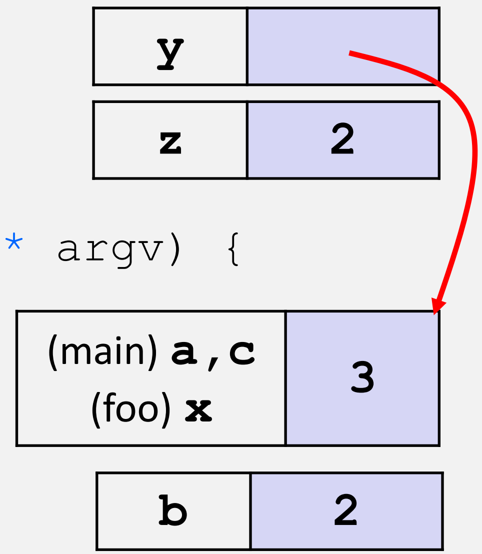
**E. We're lost…**

```
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);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```



# Poll Everywhere

**pollev.com/tqm**

- What will happen when we run this?

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

A. Output "(1,2,3)"

B. Output "(3,2,3)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

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);
→ std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```