C++ References, Const, Classes
CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:
Audrey Seo
Deeksha Vatwani
Derek de Leuw
Katie Gilchrist

Administrivia

- Exercise 8 due this morning
- No new exercise today – get ahead on hw2; longer exercise coming Friday, due Monday morning
- Exercise grades through exercise 6 are posted

Administrivia

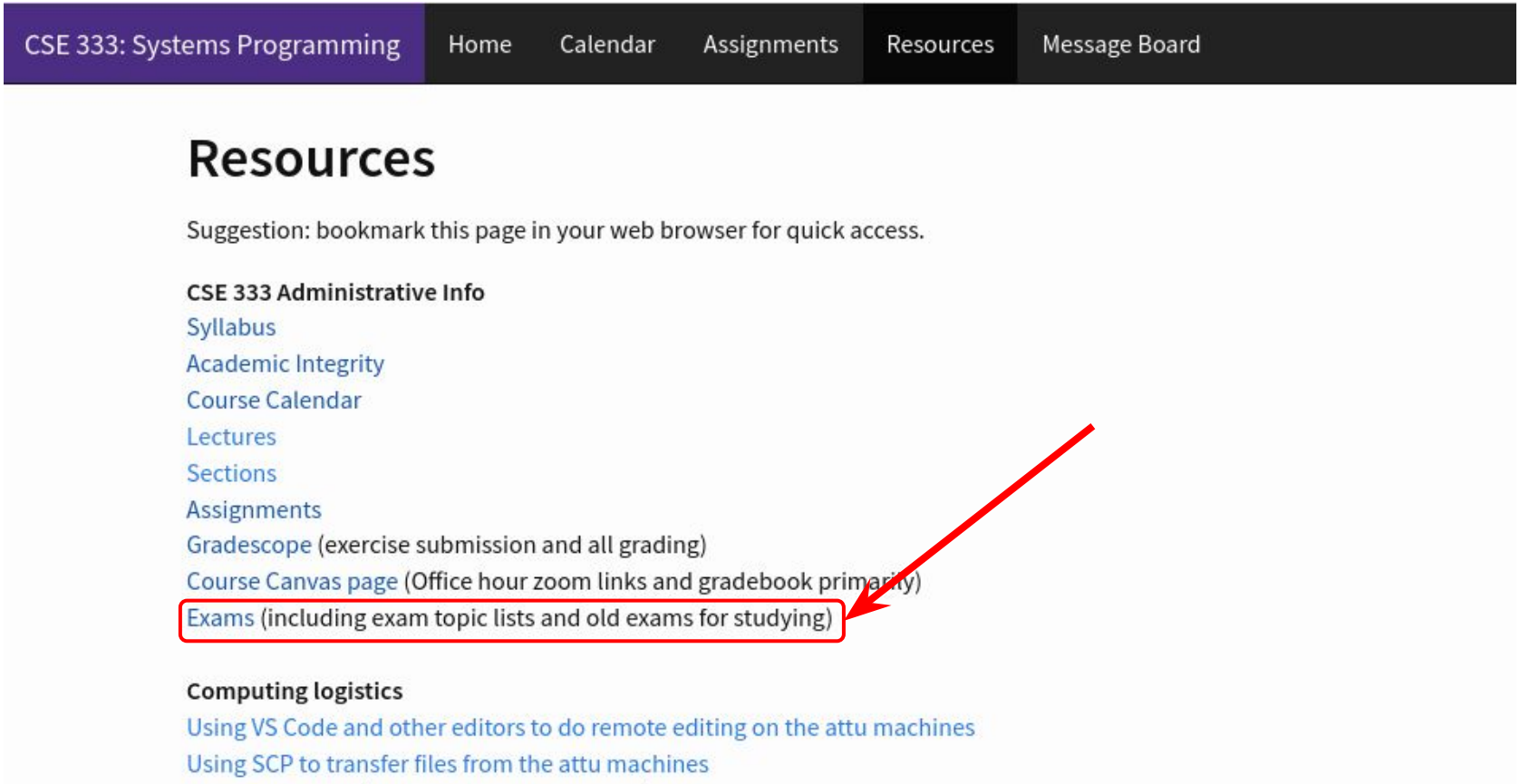
- Sections this week: the material covered in lecture today plus **Makefiles**
  - Makefiles will not be covered in lecture
  - Makefiles will be required for most exercises going forward
  - Makefiles will probably be on the midterm
  - Go to section!

Administrivia

- Homework 2 due next Thursday (7/18)
  - Note: `libhw1.a` (yours or ours) needs to be in correct directory (`hw1/`) for hw2 to build
  - Use Ctrl-D (eof) on a line by itself to exit `searchshell`; must free all allocated memory
  - Test on directory of small self-made files where you can predict the data structures and then check them
  - Valgrind takes a *long* time on the full test_tree. Try using enron docs only or other small test data directory for quick checks.

Midterm Exam 7/28 1:10 - 2:10 HRC 155

- Midterm topics and old exams are posted
  - Available on the "Resources" page



Midterm Exam 7/28 1:10 - 2:10 HRC 155

- Midterm topics and old exams are posted
  - Available on the "Resources" page
  - Closed book, slides, etc., but you may have one 5x8 notecard with whatever handwritten notes you want on both sides
  - Reference sheets with the declarations of useful functions will be included on exam

Midterm Exam 7/28 1:10 - 2:10 HRC 155

- Extra midterm points for coming to office hours next week
  - +5 points on the midterm (out of 100), but can't go above 100 total
  - Must go to an existing, in-person office hours and bring a problem set to work on; either from the extra-problems in the slides, or an old midterm question
  - Make sure the TA writes down your name and netid

Lecture Outline

- **C++ References**
- `const` in C++
- C++ Classes Intro

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++

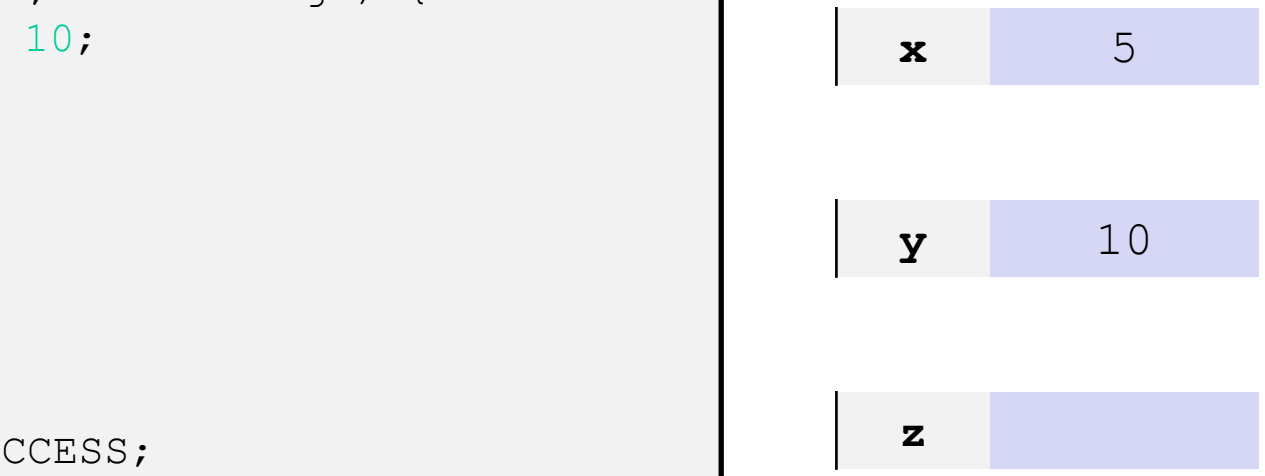
```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;

  *z += 1;
   x += 1;

   z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```

pointer.cc



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++

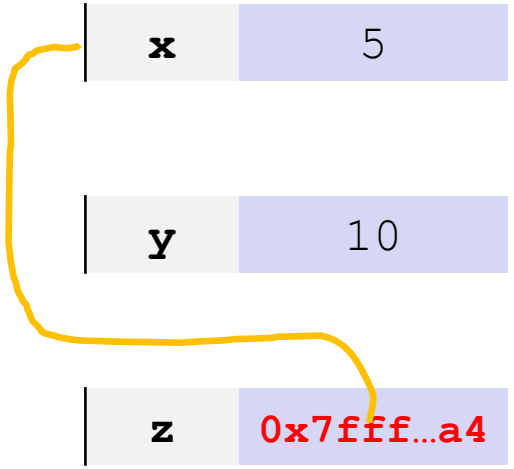
```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;

  *z += 1;
   x += 1;

   z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```

pointer.cc



Pointers Reminder

Note: Arrow points to *next* instruction.

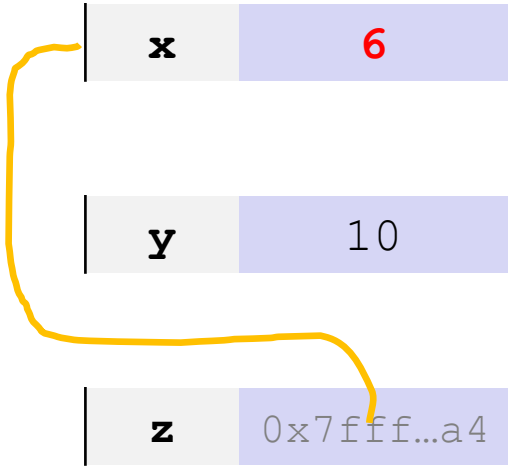
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

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

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

   z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



pointer.cc

Pointers Reminder

Note: Arrow points to *next* instruction.

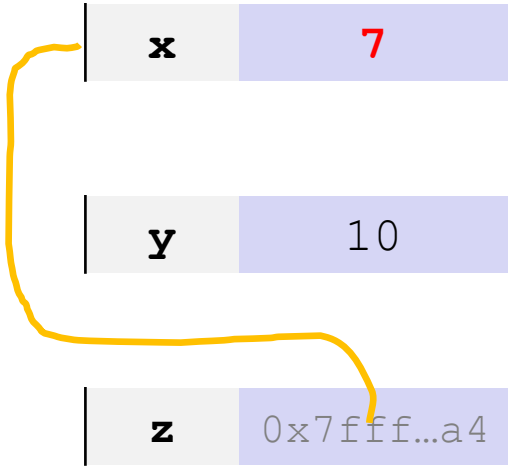
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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;
  *z += 1;

  return EXIT_SUCCESS;
}
```



pointer.cc

Pointers Reminder

Note: Arrow points to *next* instruction.

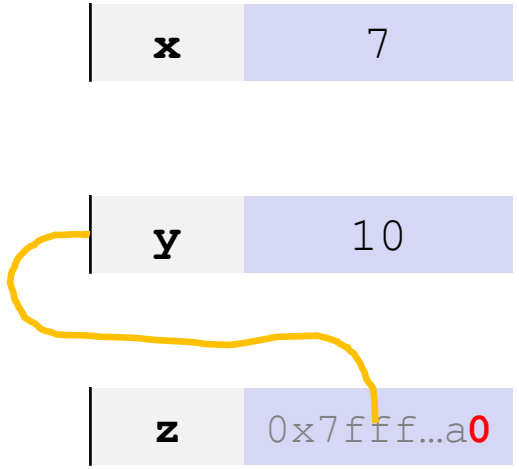
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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;
}
```



pointer.cc

Pointers Reminder

Note: Arrow points to *next* instruction.

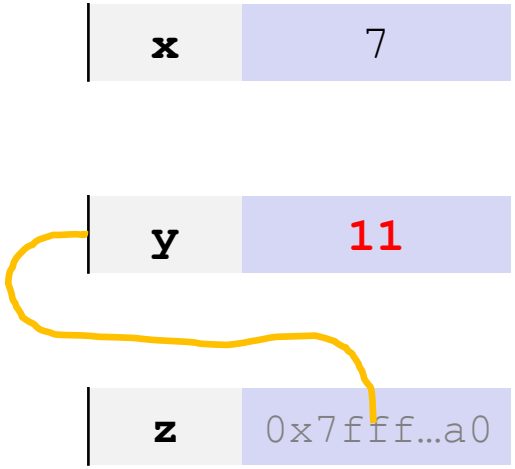
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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;
}
```



pointer.cc

References

- A **reference** is a direct alias for another 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;
}
```

reference.cc

Comparing our Examples

- 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;
}
```

```
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;
}
```

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;
}
```

x	5

y	10

reference.cc

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

reference.cc

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

reference.cc

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;
  z += 1;

  return EXIT_SUCCESS;
}
```

x, z	7

y	10

You can't rebind a reference! You can only bind it when it's declared

reference.cc

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

reference.cc

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

reference.cc

References: Be Careful!

- The "&" character is used for multiple things in C++!
  - When to the left of a **value**, it means "take the address of <value>"

```
x = &y;  // sets x to the address of y
```

OR

```
int* z = &y;  // sets z to the address of y
```

  - When to the right of a **type**, it means "reference to <type>"

```
int& x = y;  // declares x as a reference to y
```

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;
}
```

Be careful! As a client you might not expect the arguments to change.

passbyreference.cc

Pass-By-Reference

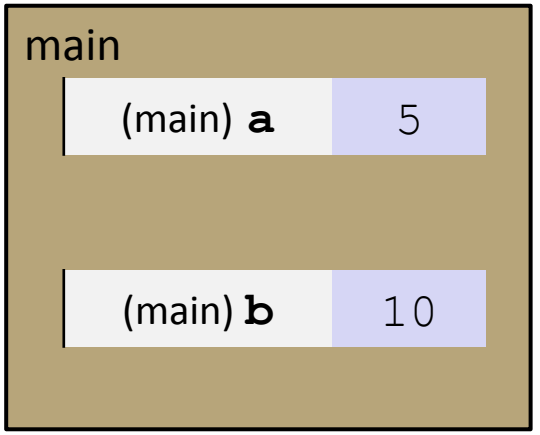
Note: Arrow points to *next* instruction.

- 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;
}
```



passbyreference.cc

Pass-By-Reference

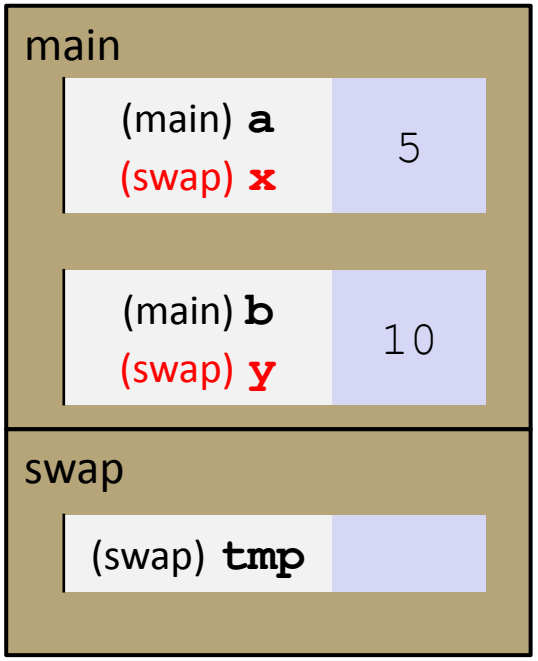
Note: Arrow points to *next* instruction.

- 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;
}
```



passbyreference.cc

Pass-By-Reference

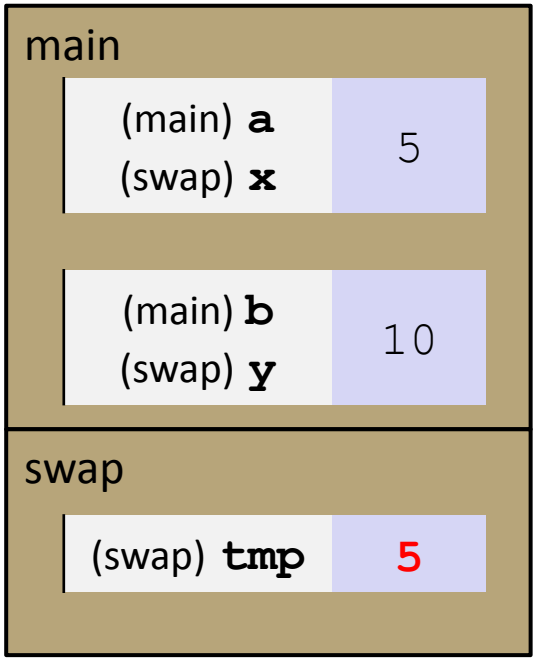
Note: Arrow points to *next* instruction.

- 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;
}
```



passbyreference.cc

Pass-By-Reference

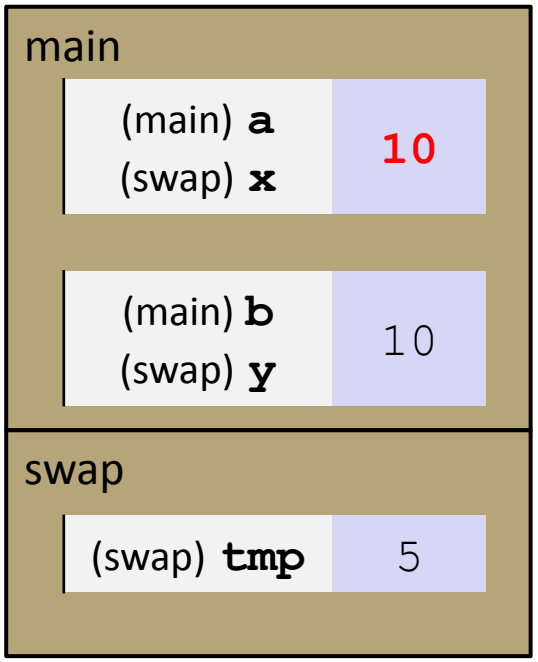
Note: Arrow points to *next* instruction.

- 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;
}
```



passbyreference.cc

Pass-By-Reference

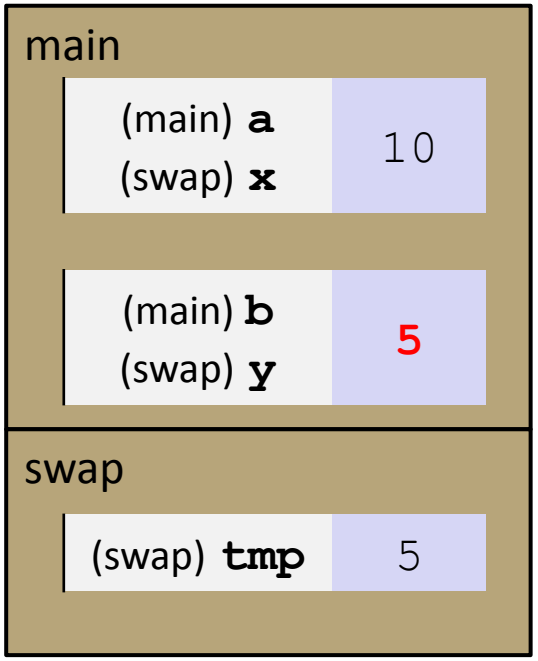
Note: Arrow points to *next* instruction.

- 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;
}
```



passbyreference.cc

Pass-By-Reference

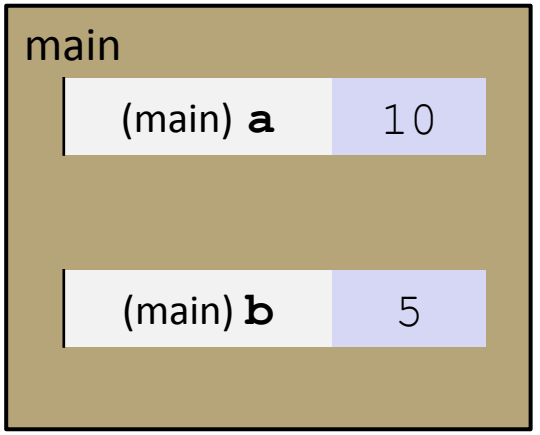
Note: Arrow points to *next* instruction.

- 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;
}
```



passbyreference.cc

- At this point, which addresses are identical? In other words: which pairs of names are aliases?
  - `&a == &b`
  - `&a == &x`
  - `&y == &tmp`

```
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;
}
```

passbyreference.cc

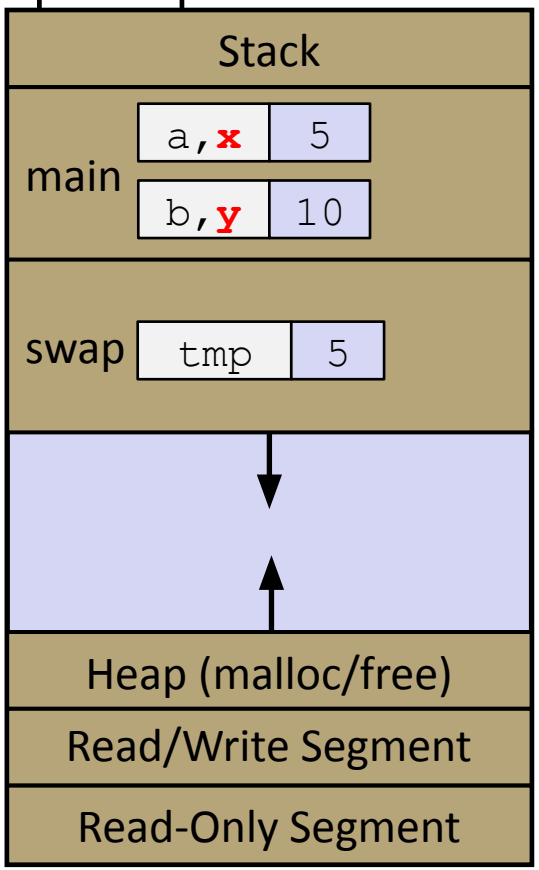
Pass-By-Reference: Mental Model

- A **reference** is an alias for another variable
  - … so it's as if no additional space is allocated for it
  - Unlike a pointer, which **is** a variable and **does** require space

```
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;
}
```



passbyreference.cc

Lecture Outline

- C++ References
- **`const` in C++**
- C++ Classes Intro

const

- `const`: this cannot be changed/mutated
  - Produces a compile-time error if you try to change it
  - Signal of intent to compiler; meaningless at hardware level

```
void BrokenPrintSquare(const int& i) {
  i = i*i; // compiler error here!
  std::cout << i << std::endl;
}

int main(int argc, char** argv) {
  int j = 2;
  BrokenPrintSquare(j);
  return EXIT_SUCCESS;
}
```

brokenpassbyrefconst.cc

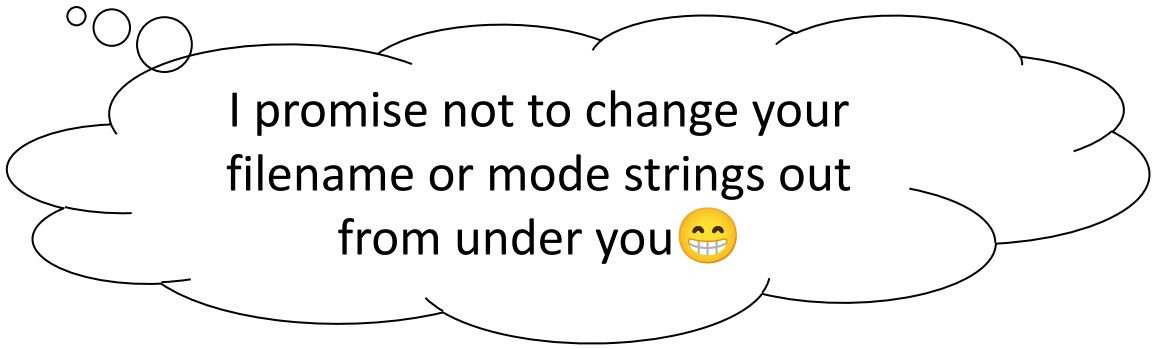
const

- Exists in C too
  - But used *much* more in C++

```
FILE* fopen(char* filename, char* mode);
```

Is actually

```
FILE* fopen(const char* filename, const char* mode);
```



`const` and Pointers

- Since pointers are variables, they can be used to modify a program's state by:
  1) Changing the value of the pointer (what it points to)
  2) Changing the thing the pointer points to (via dereference)

Pointers Reminder

Note: Arrow points to *next* instruction.

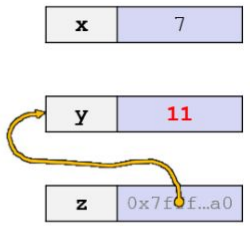
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
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;
}
```



pointer.cc

13

`const` and Pointers

- Since pointers are variables, they can be used to modify a program's state by:
  1) Changing the value of the pointer (what it points to)
  2) Changing the thing the pointer points to (via dereference)
- `const` can be used to prevent either/both of these behaviors!
  - `const` next to pointer name means you can't change the value of the pointer
  - `const` next to data type pointed to means you can't use this pointer to change the thing being pointed to

const and Pointers

- The syntax with pointers is confusing:

```
int main(int argc, char** argv) {
  int x = 5;                  // int
  const int y = 6;            // (const int)
  y++;                        // compiler error

  const int *z = &y;          // pointer to a (const int)
  *z += 1;                    // compiler error
  z++;                        // ok

  int *const w = &x;          // (const pointer) to a (variable int)
  *w += 1;                    // ok
  w++;                        // compiler error

  const int *const v = &x;    // (const pointer) to a (const int)
  *v += 1;                    // compiler error
  v++;                        // compiler error

  return EXIT_SUCCESS;
}
```

constmadness.cc

const Parameters

- A `const` parameter *cannot* be mutated inside the function
  - Therefore it does not matter if the argument is `const` (can be mutated) or not
- A non-`const` parameter *could* be mutated inside the function
  - It would be BAD if you could pass it a `const` var
  - Illegal regardless of whether *or not* the function actually tries to change the var

```
void foo(const int* y) {
  std::cout << *y << std::endl;
}

void bar(int* y) {
  std::cout << *y << std::endl;
}

int main(int argc, char** argv) {
  int a = 10;
  const int b = 20;

  foo(&a);    // OK
  foo(&b);    // OK

  bar(&a);    // OK
  bar(&b);    // not OK - error

  return EXIT_SUCCESS;
}
```

Google Style Guide Convention

- Use `const` references or call-by-value for input values
  - Particularly for large values, use references (no copying)
- Use const pointers for output parameters
- List input parameters first, then output parameters

An ordinary int (not int&) would probably be better here, since it's small, but this shows how const refs can be used

```
void CalcArea(const int& width, const int& height,
              int* const area) {
  *area = width * height;
}
```

styleguide.cc

Lecture Outline

- C++ References
- `const` in C++
- **C++ Classes Intro**

Classes

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

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

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

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

Class Member Functions

- Class member functions can be:

1. *Declared* within the class definition and then *defined* elsewhere

```
class Name {
  retType MethodName(type1 param1, …, typeN paramN);
};  // class Name
```

.h file

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

.c file

2. *Defined* within the class definition
   - typically only used for trivial method definitions, like getters/setters

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

.h file

Class Organization (.h/.cc)

- It's a little more complex than in C when modularizing with `struct` definition:
  - Class definition is part of interface and should go in `.h` file
    - Private members still must be included in definition (!)
  - Usually put member function definitions into companion `.cc` file with implementation details
  - These files can also include non-member functions that use the class (more about this later)
- Unlike Java, you can name files anything you want
  - But normally `Name.cc` and `Name.h` for `class Name`

Class Definition (.h file)

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(const int x, const int y);        // constructor
  int get_x() { return x_; }               // inline member function
  int get_y() { 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_
```

These are defined

Everything under here is private

These are just declared

Class Member Definitions (. cc file)

Point.cc

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

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

double Point::Distance(const Point& p) const {
  // 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(const int x, const int y) {
  x_ = x;
  y_ = y;
}
```

The method name is the class name for constructors

Use ClassName::MethodName when defining

Class Usage (a different .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
  Point p2(4, 6);  // allocate a new Point on

  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;
}
```

You can break your prints into many lines

Class Usage (a different .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
  const 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() << ", ";  // Compiler error
  cout << p2.get_y() << ")" << endl;        // Compiler error

  cout << "dist : " << p1.Distance(p2) << endl;
  return 0;
}
```

Class Definition (.h file)

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(const int x, const int y);      // constructor
  int get_x() { return x_; }            // inline member function
  int get_y() { 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_
```

Class Definition (.h file)

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(const int x, const int y);      // constructor
  int get_x() const { return x_; }      // inline member function
  int get_y() const { return y_; }      // inline member function
  double Distance(const Point& p) const;      // 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_
```

Reading Assignment

- Before next time, you **must *read*** the sections in *C++ Primer* covering class constructors, copy constructors, assignment (`operator=`), and destructors
  - Ignore "move semantics" for now
  - The table of contents and index are your friends…

Reading Assignment

- Go to
  https://orbiscascade-washington.primo.exlibrisgroup.com/permalink/01ALLIANCE_UW/1juclfo/alma99162157785001452
  - You can also search the uw library for "C++ Primer Plus"
- Click on the link to O'Reilly Academic.
- Use the table of contents to navigate to
  - 10.7-10.9
    - "Declaring and Defining Constructors"
    - "Using Constructors"
    - "Default Constructors"
  - 12.2-12.4
    - "Special Member Functions"
    - "Back to Stringbad: …"
    - "More Stringbad Problems: …"

Reading Assignment

- Seriously – the next lecture will make a ***lot*** more sense if you've done some background reading ahead of time
  - Don't worry whether it all makes sense the first time you read it – it won't! The goal is to be aware of what the main issues are….
- There will be a poll at the beginning of class on what you've read

Makefiles

- Don't forget to go to section tomorrow to learn about Makefiles!

Extra Exercise #1

- Write a C++ program that:
  - Has a class representing a 3-dimensional point
  - Has the following methods:
    - Return the inner product of two 3D points
    - Return the distance between two 3D points
    - Accessors and mutators for the `x`, `y`, and `z` coordinates

Extra Exercise #2

- Write a C++ program that:
  - Has a class representing a 3-dimensional box
    - Use your Extra Exercise #1 class to store the coordinates of the vertices that define the box
    - Assume the box has right-angles only and its faces are parallel to the axes, so you only need 2 vertices to define it
  - Has the following methods:
    - Test if one box is inside another box
    - Return the volume of a box
    - Handles <<, =, and a copy constructor
    - Uses `const` in all the right places