

pollev.com/cse333

About how long did Exercise 4 take you?

- A. [0, 2) hours
- B. [2, 4) hours
- C. [4, 6) hours
- D. [6, 8) hours
- E. 8+ Hours
- F. I didn't submit / I prefer not to say

Systems Programming

C++ References, Const, Classes

Instructor: Justin Hsia

Teaching Assistants:

Aadi Adgaonkar

Advaith Vijayakumar

Alice Fu

Andrew Vlahovich

Aviel Wood

Camden Harris

Esha Gore

Hannah Hempstead

Helena Wang

Laura Wang

Megan Frisella

Nina Tharamal

Shrihun Sankepally

Shriya Sateesh

Yusong Yan

Relevant Course Information

- ❖ Exercise 5 due Friday
- ❖ Homework 2 due a week from Thursday (10/29)
 - File system crawler, indexer, and search engine (Ctrl-D to exit searchshell)
 - Test on a directory of small self-made files; draw out what you expect the data structures should look like
 - “Check-in 1 (HW2)” due Tuesday, 10/27
- ❖ Midterm is Monday, November 2 from 5:30–6:40 PM
 - Located in [GUG 220](#) or [ARC 147](#), depending on your quiz section
 - Ed post with details forthcoming
 - Practice midterms and solutions on the course website
 - Midterm review (hybrid) from 4:30–6:20 PM

Lecture Outline (1/3)

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

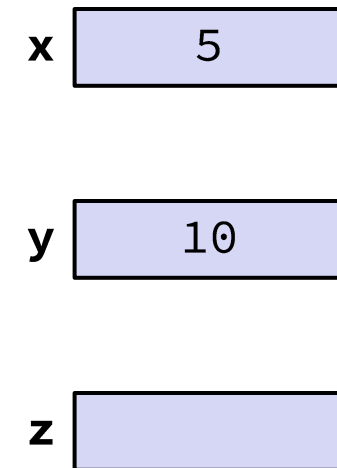
Pointers Reminder (1/6)

Note: arrow points to *next* instruction to be executed.

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify the variable 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

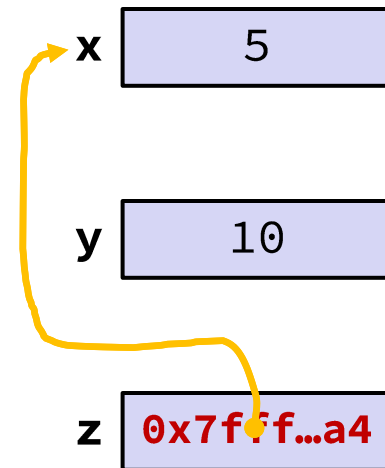


Pointers Reminder (2/6)

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify the variable 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

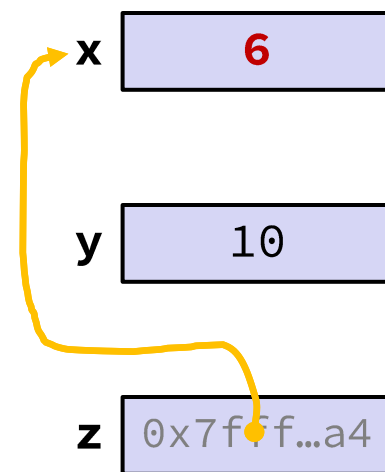


Pointers Reminder (3/6)

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify the variable 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

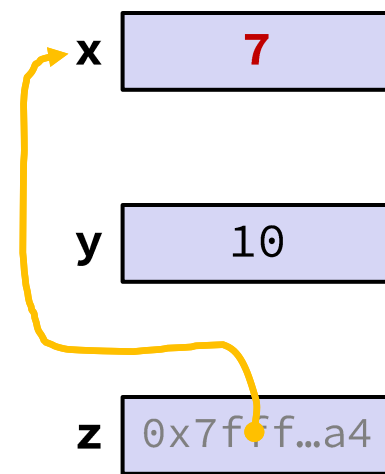


Pointers Reminder (4/6)

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify the variable 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

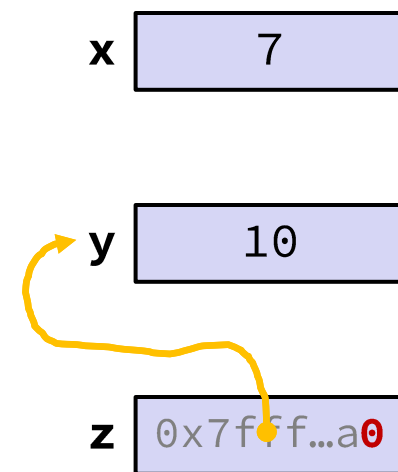


Pointers Reminder (5/6)

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify the variable 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



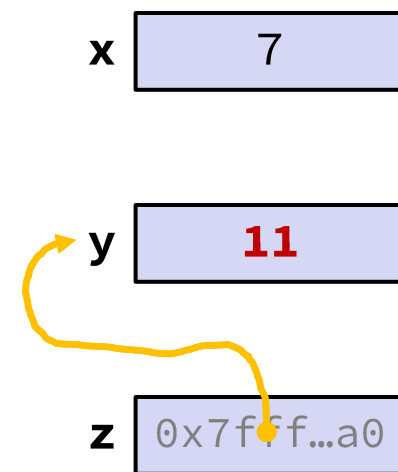
Pointers Reminder (6/6)

- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify the variable 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



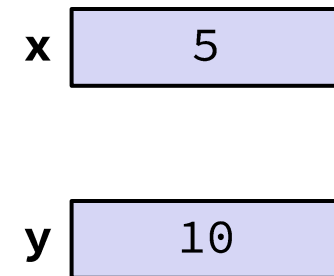
References (1/6)

Note: arrow points to *next* instruction to be executed.

- ❖ 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;  
}
```

reference.cc

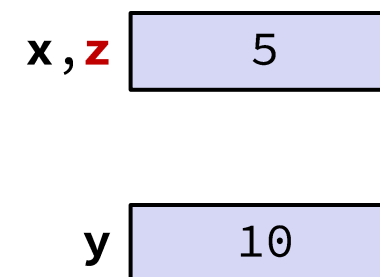


References (2/6)

- ❖ 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;  
}
```

reference.cc



References (3/6)

- ❖ 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

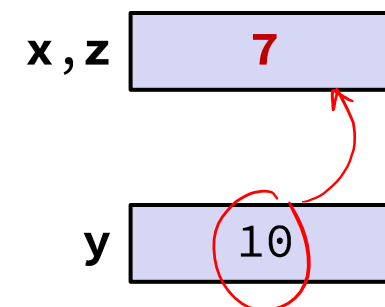
reference.cc

References (4/6)

- ❖ 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;  
}
```

reference.cc



References (5/6)

- ❖ 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

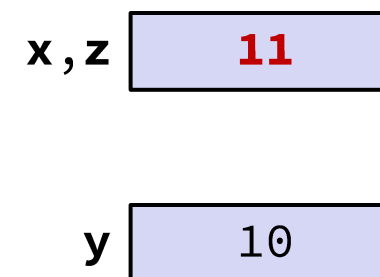
reference.cc

References (6/6)

- ❖ 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;  
}
```

reference.cc



Pass-By-Reference (1/6)

Note: arrow points to *next* instruction to be executed.

- ❖ 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;  
}
```

aliases that are bound to arguments

→

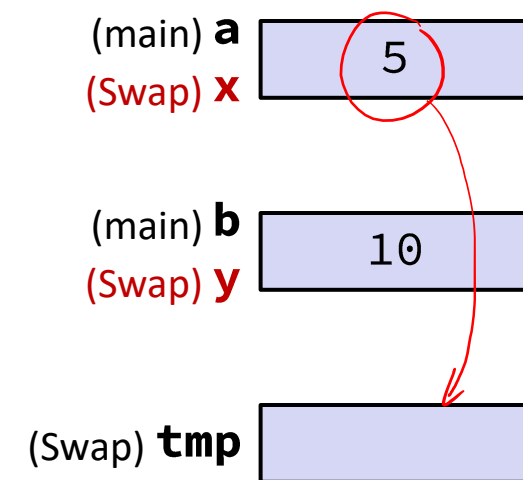
(main) **a** 5

(main) **b** 10

Pass-By-Reference (2/6)

- ❖ 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!

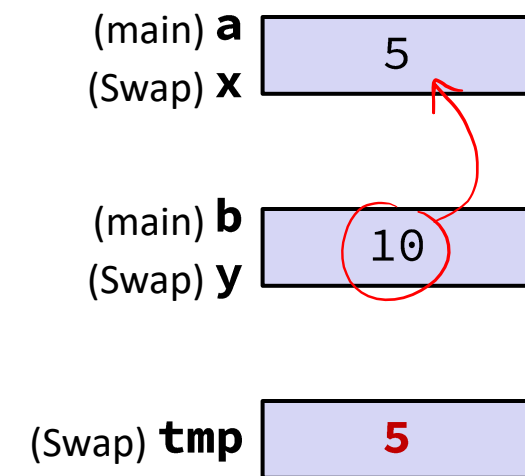
```
void Swap(int& x, int& y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 5, b = 10;  
    Swap(a, b);  
    cout << "a: " << a << "; b: " << b << endl;  
    return EXIT_SUCCESS;  
}
```



Pass-By-Reference (3/6)

- ❖ 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!

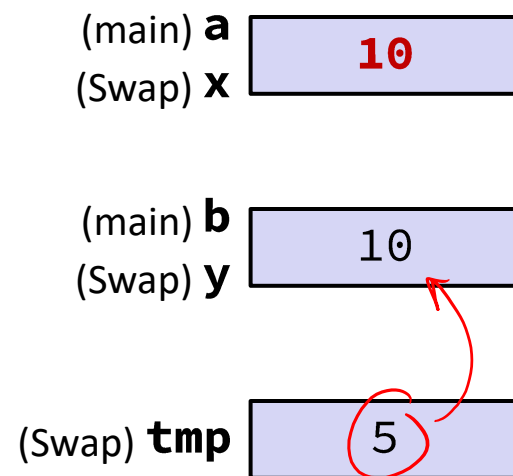
```
void Swap(int& x, int& y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 5, b = 10;  
  
    Swap(a, b);  
    cout << "a: " << a << "; b: " << b << endl;  
    return EXIT_SUCCESS;  
}
```



Pass-By-Reference (4/6)

- ❖ 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 (5/6)

- ❖ 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
(Swap) **x**

(main) **b** 5
(Swap) **y**

(Swap) **tmp** 5

Pass-By-Reference (6/6)

- ❖ 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

pollev.com/cse333

What will happen when we try to compile and run this code? (1)

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

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

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

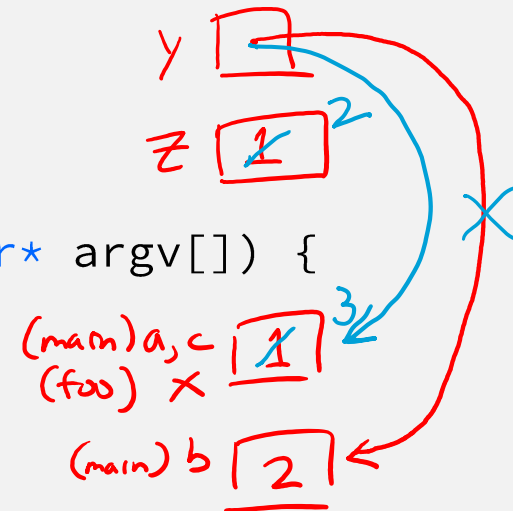
D. Compiler error about body of Foo

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;  
}
```

poll1.cc



Lecture Outline (2/3)

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

const

❖ **const**: this cannot be changed/mutated

- Used *much* more in C++ than in C

- Signal of intent to compiler; meaningless at hardware level
 - Results in compile-time errors

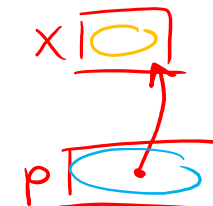
```
void BrokenPrintSquare(const int& i) {  
    i = i*i; // compiler error here!  
    std::cout << i << std::endl;  
}  
  
int main(int argc, char* argv[]) {  
    int j = 2;  
    BrokenPrintSquare(j);  
    return EXIT_SUCCESS;  
}
```

brokenpassbyrefconst.cc

const and Pointers

❖ Pointers can change data in two different contexts:

`int x;`
`int *p = &x;`



- 1) You can change the value of the pointer (*i.e.*, address stored)
- 2) You can change 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
 - e.g., `int* const p;` X ✓
- `const` next to data type pointed to means you can't use this pointer to change the thing being pointed to
 - e.g., `const int* p;` ✓ X
- Tip: read variable declaration from *right-to-left*

const and Pointers Examples

yellow = deref
blue = address

❖ The syntax with pointers is confusing:

```
int main(int argc, char* argv[]) {  
    int x = 5; // int  
    const int y = 6; // (const int)  
    ✗ ++y;  
    const int* z = &y; // pointer to a (const int)  
    ✗ *z += 1;  
    ✓ ++z;  
    int* const w = &x; // (const pointer) to a (variable int)  
    ✓ *w += 1;  
    ✗ ++w;  
    const int* const v = &x; // (const pointer) to a (const int)  
    ✗ *v += 1;  
    ✗ ++v;  
    return EXIT_SUCCESS;  
}
```

constmadness.cc

const and Pointers Examples: Outcomes

❖ 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 whether the argument can be mutated or not
- ❖ A non-const parameter *may* be mutated inside the function
 - Compiler won't let you pass in const parameters
- ❖ Make parameters const when you can!

```
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[]) {  
    const int a = 10;  
    int b = 20;  
  
    { Foo(&a); // OK  
      Foo(&b); // OK  
      Bar(&a); // not OK - error  
      Bar(&b); // OK  
    }  
  
    return EXIT_SUCCESS;  
}
```

Handwritten notes in red:

- Under `const int*` in `Foo`: const int*
- Arrow from `int*` in `Bar` to `*y` in `Bar`: *doesn't actually modify value of y!*
- Arrow from `Bar(&a);` to the list item "Compiler won't let you pass in const parameters": *not OK - error*

pollev.com/cse333

What will happen when we try to compile and run this code? (2)

can't change stored addr, but can change what it points to

local copy of int

A. Output "(2,4,0)"

B. Output "(2,4,3)"

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

D. Compiler error about body of Foo

E. We're lost...

```
void Foo(int* const x, int& y, int z) {  
    *x += 1; // allowed  
    y *= 2; // allowed  
    z -= 3; // allowed, but has no lasting effect  
}  
  
int main(int argc, char* argv[]) {  
    const int a = 1;  
    int b = 2, c = 3;  
    Foo(&a, b, c);  
    std::cout << "(" << a << "," << b  
        << "," << c << ")" << std::endl;  
    return EXIT_SUCCESS;  
}
```

poll2.cc



When to Use References?

- ❖ A stylistic choice, not mandated by the C++ language
- ❖ Google C++ Style Guide suggests:
 - Input parameters:
 - Values for primitive types like `int` or small structs/objects
 - `const` references for complex struct/object instances
 - Output parameters:
 - Use `const` pointers (unchangeable pointers referencing changeable data)
 - List input parameters first, then output parameters last

```
size_t StringBefore(const string& input, const string& search,
                    string* const result) {
    *result = input.substr(0, input.find(search));
    return input.find(search);
}
```

styleguide.cc

constant pointer to mutable string

Lecture Outline (3/3)

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

Classes

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

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

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

↖ don't forget!

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

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

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

Class Organization

- ❖ 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
 - Common exceptions: setter and getter methods
 - These files can also include **non-member functions** that use the class
- ❖ Unlike Java, you can name files anything you want
 - Typically `Name.cc` and `Name.h` for **class** **Name**

const & Classes

- ❖ Like other data types, **objects** can be declared as `const`:
 - Once a `const` object has been constructed, its member variables can't be changed
 - Can only invoke member functions that are labeled `const`

- ❖ You can declare a member **function** of a class as `const`
 - This means that it cannot modify the object it was called on
 - The compiler will treat member variables as `const` inside the function at compile time
 - If a member function doesn't modify the object, mark it `const`!



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_
```

declarations

this const means that this function is not allowed to change the object on which it is called (the implicit "this" pointer)

function definitions

compiler may choose to expand inline (like a macro) instead of an actual function call

naming convention for class data members (Google C++ style guide)

Class Member Definitions (.cc file)

Point.cc

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

Point::Point(const int x, const int y) {
    // BAD STYLE to mix-and-match "this->var_" and "var_"
    x_ = x;
    this->y_ = y; // "this->" is optional unless name conflicts
}

double Point::Distance(const Point& p) const {
    // In a member function of the same class, can access p's x_
    // and y_ members directly or using the get_x()/get_y()
    // accessor functions (BAD STYLE to mix-and-match, though).
    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;
}
```

Handwritten annotations:

- Red arrow from `equivalent to  $y_- = y;$` points to `this->y_ = y;`
- Red arrow from `"this" is a (Point* const)` points to `this->`
- Red arrow from `makes "this" a (const Point* const)` points to `const` in `Distance`
- Red arrow from `equivalent to  $p.x_-$` points to `p.get_x()`
- Red arrow from `can't be const because we are mutating "this"` points to `SetLocation`

Class Usage (.cc file)

```
#include "Point.h"

#include <iostream>
#include <cstdlib>

using std::cout;
using std::endl;

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 EXIT_SUCCESS;
}
```

usepoint.cc

} calls defined
constructor

"dot notation" used for member functions

Reading Assignment

- ❖ Before next time, **read** the sections in *C++ Primer* covering class constructors, copy constructors, assignment (operator=), and destructors
 - Free link for UW students to the [relevant section \(§13.1.1\)](#)
 - Click “Read now” and scroll down to “ Academic user?”
 - Ignore “move semantics” for now
 - The table of contents and index are your friends...