

C++ References, Const, Classes

CSE 333 Summer 2020

Instructor: Travis McGaha

Teaching Assistants:

Jeter Arellano

Ian Hsiao

Ramya Challa

Allen Jung

Kyrie Dowling

Sylvia Wang



Poll Everywhere

pollev.com/cse33320su

About how long did Exercise 8 take?

- A. 0-1 Hours
- B. 1-2 Hours
- C. 2-3 Hours
- D. 3-4 Hours
- E. 4+ Hours
- F. I didn't submit / I prefer not to say

Side question:

What do you want to ask Travis?
(Doesn't have to be 333 related)

Administrivia

- ❖ Exercise 9 released today, due Friday
 - Write a substantive class in C++ (but no dynamic allocation – yet)
 - First submitted Makefile!
- ❖ Homework 2 due next Thursday (7/23)
 - File system crawler, indexer, and search engine
 - Note: libhw1.a (yours or ours) and the .h files from hw1 need to be in right directory (~yourgit/hw1/)
 - Note: use Ctrl-D to exit searchshell, test on directory of small self-made files

Lecture Outline

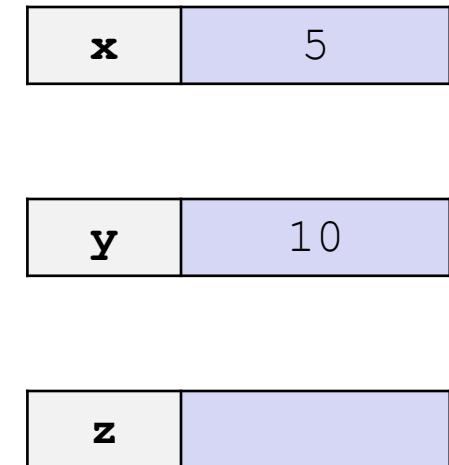
- ❖ C++ References
- ❖ const in C++
- ❖ C++ Classes Intro

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;  
}
```

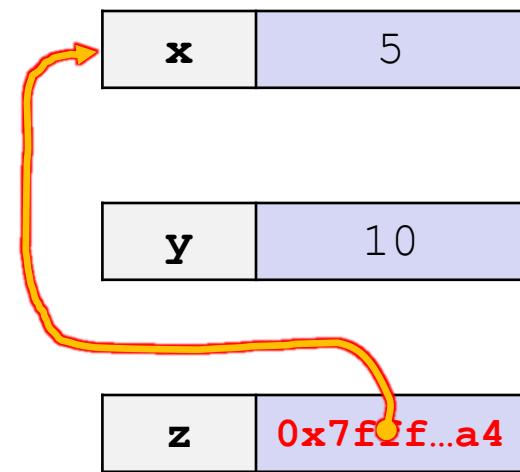


Pointers Reminder

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;  
}
```

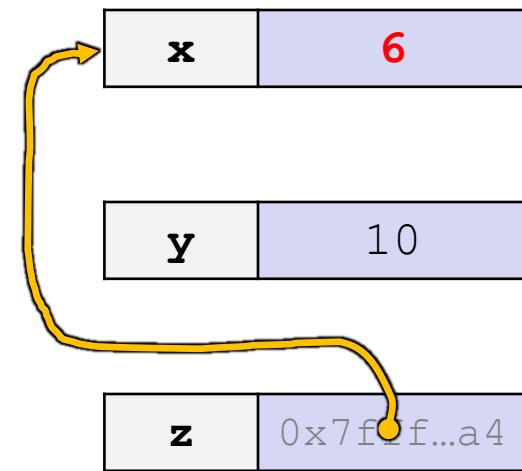


Pointers Reminder

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; // sets x to 6  
    x += 1;  
  
    z = &y;  
    *z += 1;  
  
    return EXIT_SUCCESS;  
}
```

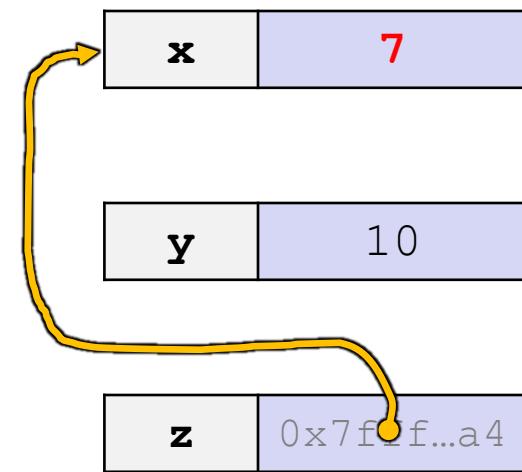


Pointers Reminder

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;    // sets x to 6  
    x += 1;    // sets x (and *z) to 7  
  
    z = &y;  
    *z += 1;  
  
    return EXIT_SUCCESS;  
}
```

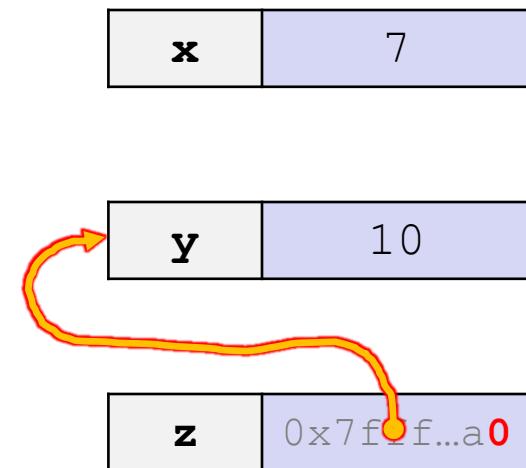


Pointers Reminder

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;  
}
```

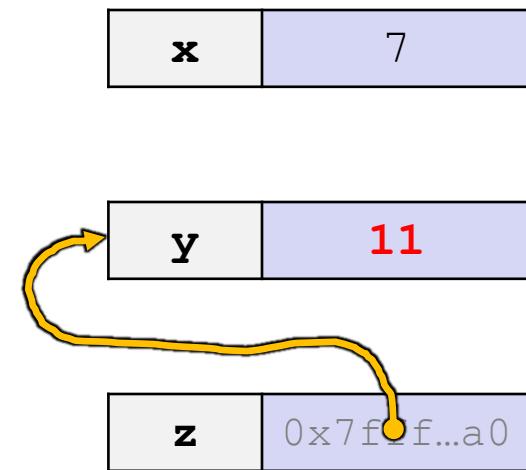


Pointers Reminder

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;  
}
```

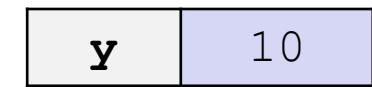
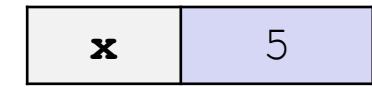


References

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;           // When we use '&' in a type  
    x += 1;           declaration, it is a reference.  
  
    z = y;           &var still is "address of var"  
    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;  
    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;  
}
```



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
----------	---

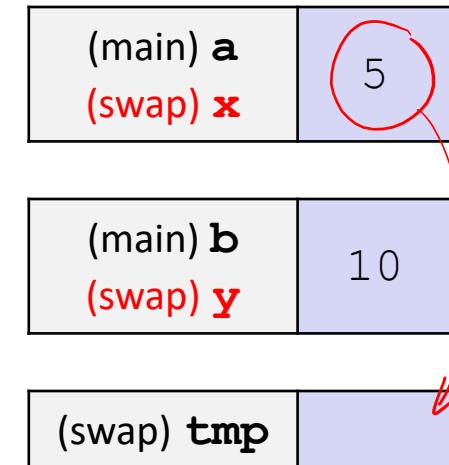
(main) b	10
----------	----

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;  
}
```

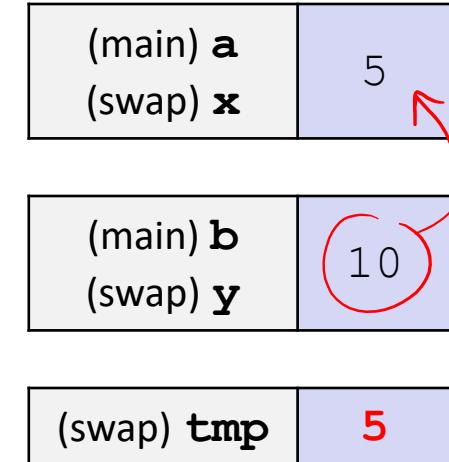


Pass-By-Reference

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;  
}
```



(main) a	10
(swap) x	



(main) b	10
(swap) y	



(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
(swap) x	

(main) b	5
(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





- ❖ 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...

poll1.cc

```
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/cse33320su

- ❖ 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...

poll1.cc

```
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



pollev.com/cse33320su

- ❖ What will happen when we run this?

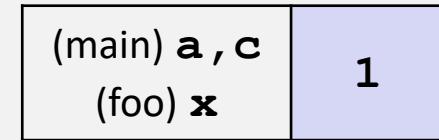
Note: Arrow points to *next* instruction.

- 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...

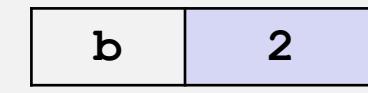
```
void foo(int& x, int* y, int z) {  
    →z = *y;  
    x += 2;  
    y = &x;  
}
```



```
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/cse33320su

- ❖ What will happen when we run this?

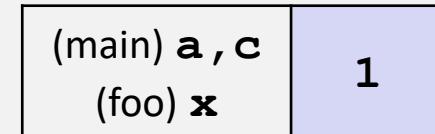
Note: Arrow points to *next* instruction.

- Output "(1,2,3)"
- Output "(3,2,3)"
- Compiler error about arguments to foo (in main)
- Compiler error about body of foo
- We're lost...

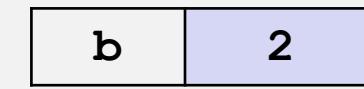
```
void foo(int& x, int* y, int z) {
    z = *y;
    →x += 2;
    y = &x;
}
```



```
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

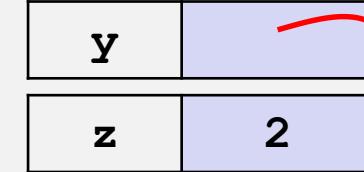
pollev.com/cse33320su

- ❖ What will happen when we run this?

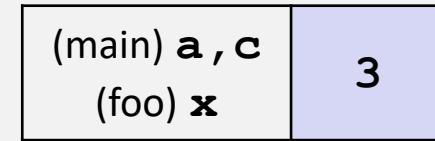
Note: Arrow points to *next* instruction.

- 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...

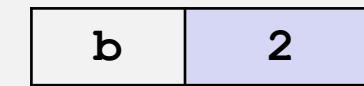
```
void foo(int& x, int* y, int z) {  
    z = *y;  
    x += 2;  
    → y = &x;  
}
```



```
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;
```

```
}
```



pollev.com/cse33320su

- ❖ 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) {
```

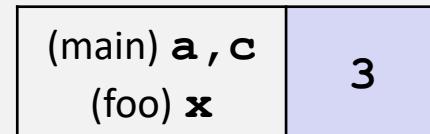
```
    z = *y;  
    x += 2;  
    y = &x;
```

```
}
```



```
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;
```

```
}
```

poll1.cc



Poll Everywhere

pollev.com/cse33320su

- ❖ 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...

poll1.cc

```
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	3
-----	---

b	2
---	---

Lecture Outline

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

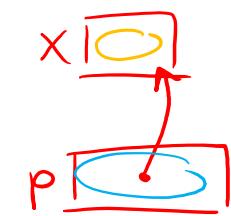
const

- ❖ `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:
 - 1) You can change the value of the pointer $\text{int } x;$ $\text{int } *p = \&x;$ 
 - 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 $\text{int } * \text{const } p;$
 - const next to data type pointed to means you can't use this pointer to change the thing being pointed to $\text{const } \text{int } *p;$
 - Tip: read variable declaration from *right-to-left*

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++;

    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;
}
```

Zoom Voting:

✓ Compiles

yes

✗ Compiler error

no

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;
}
```

const Parameters

- ❖ A **const parameter**
cannot be mutated inside the function
 - Therefore it does not matter if 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

```
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;
}
```



- ❖ What will happen when we try to compile and run?

- 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...

poll2.cc

```
void foo(int* const x,
          int& y, int z) {
    *x += 1;
    y *= 2;
    z -= 3;
}

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;
}
```



Poll Everywhere

pollev.com/cse33320su

- ❖ What will happen when we try to compile and run?

- 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...

Can't modify the x, but can
modify *x (dereference) poll2.cc

```
void foo(int* const x,  
        int ref → int& y, int z) {  
    *x += 1; → Allowed      Copy of int value  
    y *= 2; →  
    z -= 3; →  
}  
→ Allowed, but change doesn't persist out
```

```
int main(int argc, char** argv) {  
    const int a = 1;  
    int b = 2, c = 3;  
    foo(&a, b, c); → Const mismatch  
    std::cout << "(" << a << ", " << b  
    << ", " << c << ")" << std::endl;  
  
    return EXIT_SUCCESS;  
}
```



When to Use References?

- ❖ A stylistic choice, not mandated by the C++ language
- ❖ Google C++ style guide suggests:
 - Input parameters:
 - Either use values (for primitive types like `int` or small structs/objects) *Avoid making unnecessary copies*
 - Or use `const` references (for complex struct/object instances)
 - Output parameters: *To make sure we don't change in function
also allows const & non-const arguments*
 - Use `const` pointers
 - Unchangeable pointers referencing changeable data
 - Ordering:
 - List input parameters first, then output parameters last

```
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 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 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 exception: 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
 - If a member function doesn't modify the object, mark it const
 - Compiler will treat member variables as const inside the function, and check you don't manipulate the member variables at compile time

Class Definition (.h file)



Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {           const means the object
public:                 we are calling on, can't be changed   Inline definition ok for simple
    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
```

Declarations

Google C++ naming conventions for data members

#endif // POINT_H_

Class Member Definitions (.cc file)

Point.cc

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

This code uses bad style for demonstration purposes

```
Point::Point(const int x, const int y) {
    x_ = x;           Equivalent to y_=y;
    this->y_ = y;    // "this->" is optional unless name conflicts
}                 "this" is a Point* const
double Point::Distance(const Point& p) const {Can't modify the "this" object inside the function
    // 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);
}                 const won't affect caller, but good style
void Point::SetLocation(const int x, const int y) {Can't be const. We have to mutate the Point
    x_ = x;
    y_ = y;
}
```

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)



- ❖ What will happen when we try to compile and run?

poll3.cc

- A. Output "1"
- B. Output "351"
- C. Compiler error about violating const-ness of i (in main)
- D. Compiler error about one of the member functions
- E. We're lost...

```
class Integer {  
public:  
    Integer(int x) { x_ = x; }  
    int GetValue() const { return x_; }  
    void SetValue(int x) const { x_ = x; }  
private:  
    int x_;  
};  
  
int main(int argc, char** argv) {  
    const Integer i(1);  
    i.SetValue(i.GetValue() + 350);  
    std::cout << i.GetValue() << std::endl;  
  
    return EXIT_SUCCESS;  
}
```



- ❖ What will happen when we try to compile and run?

poll3.cc

- A. Output "1"
- B. Output "351"
- C. Compiler error about violating const-ness of i (in main)
- D. Compiler error about one of the member functions
- E. We're lost...

```
class Integer {  
public:  
    Integer(int x) { x_ = x; }  
    int GetValue() const { return x_; }  
    void SetValue(int x) const { x_ = x; }  
private:  
    int x_;  
};  
  
int main(int argc, char** argv) {  
    const Integer i(1);  
    i.SetValue(i.GetValue() + 350);  
    std::cout << i.GetValue() << std::endl;  
  
    return EXIT_SUCCESS;  
}
```



- ❖ What will happen when we try to compile and run?

- A. Output "1"
- B. Output "351"
- C. Compiler error about violating const-ness of i (in main)
- D. Compiler error about one of the member functions
- E. We're lost...

poll3.cc

```
class Integer {  
public:  
    Integer(int x) { x_ = x; }  
    int GetValue() const { return x_; }  
    void SetValue(int x) const { x_ = x; }  
private:  
    int x_;  
};  
  
int main(int argc, char** argv) {  
    const Integer i(1);  
    i.SetValue(i.GetValue() + 350);  
    std::cout << i.GetValue() << std::endl;  
  
    return EXIT_SUCCESS;  
}
```

How to fix this?

Reading Assignment

- ❖ Before next time, **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...

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