Poll Everywhere

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**

C++ References, Const, Classes
CSE 333 Summer 2023

Instructor: Timmy Yang

Teaching Assistants:

Jennifer Xu   Leanna Nguyen   Pedro Amarante

Sara Deutscher   Tanmay Shah

Relevant Course Information (1/2)

- ❖ Exercise 5 due Wednesday @ 1 pm
    - ▪ "Lighter" exercise in C++ (Rating: 1)

- ❖ Homework 2 due a week from Thursday (7/20)
    - ▪ Partner sign up due tomorrow night (see Ed post #116)
    - ▪ File system crawler, indexer, and search engine
    - ▪ <u>Note</u>: `libhw1.a` (yours or ours) and the `.h` files from hw1 need to be in right directory (`~yourgit/hw1/`)
    - ▪ <u>Note</u>: use Ctrl-D to exit `searchshell`, test on directory of small self-made files

Relevant Course Information (2/2)

❖ Quiz 1 released today @ 2pm (7/10)

- Will be administered on Gradescope, closes Wednesday (7/12) @ 11:59pm
  - Quiz should take 45-30 min to complete (i.e., meant to be short).
- Please keep all Quiz questions on Ed private
  - If anything is frequently asked, we'll make a separate announcement.
- Questions about the Quiz in Office Hours can only be clarification questions.
  - TAs may ask you to post on the Ed board instead of answering directly.
- Academic Conduct Policy applies to all Quizzes as well
  - Please don't copy other's work, do not use Chat-GPT
  - https://courses.cs.washington.edu/courses/cse333/23su/quizzes/

Lecture Outline

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

Pointers Reminder

- ❖ 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++

```cpp
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;
}
```
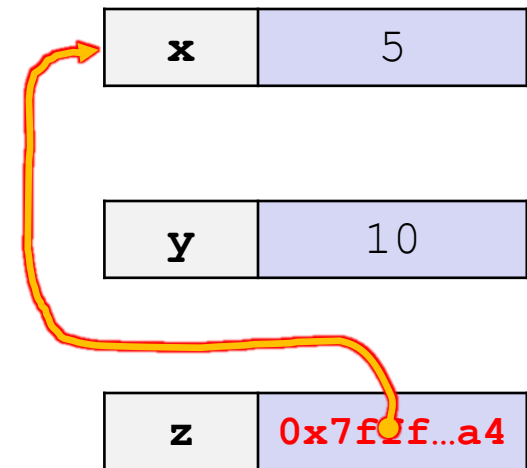
x	5

y	10

z	

pointer.cc

Pointers Reminder

❖ 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;
}
```

x	5

y	10

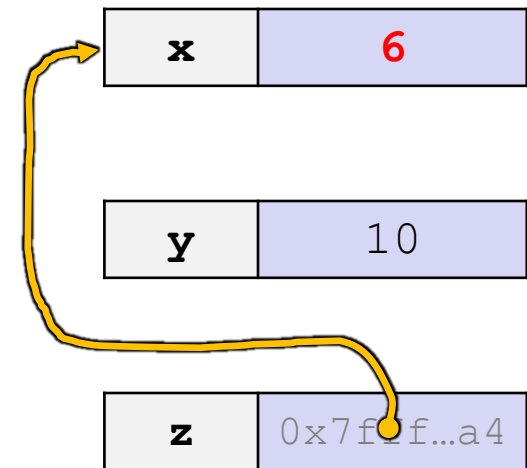| z	**0x7f2f…a4**

pointer.cc

Pointers Reminder

❖ 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;
}
```

x	6

y	10

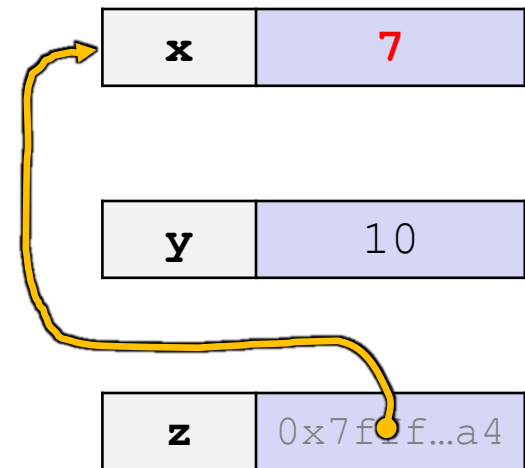| z	0x7f...f...a4

pointer.cc

8

Pointers Reminder

❖ 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;
}
```

x	7

y	10

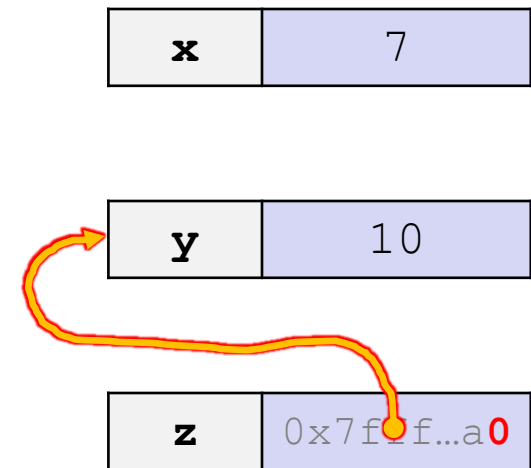| z	0x7f?f…a4

pointer.cc

Pointers Reminder

❖ 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;
}
```

x	7

y	10

z	0x7f f…a0

pointer.cc

10

Pointers Reminder

❖ 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;
}
```
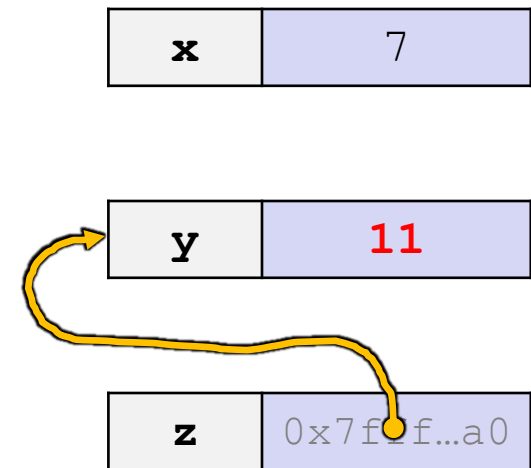
x	7

y	**11**

z	0x7f…f…a0

pointer.cc

References

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

```cpp
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

12

References

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

```cpp
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

❖ 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

```cpp
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

UNIVERSITY *of* WASHINGTON

References

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

```
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

reference.cc

References

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

```cpp
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

16

References

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

```
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

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!

```cpp
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**	5

(main) **b**	10

passbyreference.cc

18

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!

```cpp
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**	5

(main) **b** (Swap) **y**	10

(Swap) **tmp**	

passbyreference.cc

19

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!

```cpp
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**	5

(main) **b** (Swap) **y**	10

(Swap) **tmp**	**5**

passbyreference.cc

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!

```cpp
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**	10

(Swap) **tmp**	5

passbyreference.cc

21

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!

```cpp
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

passbyreference.cc

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!

```cpp
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

passbyreference.cc

23

Poll Everywhere

pollev.com/cse333

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

poll1.cc

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…**

```cpp
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;
}
```

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

```cpp
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

26

`const` and Pointers

❖ Pointers can change data in two different contexts:

1) You can change the value of the pointer

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

- `const` next to data type pointed to means you can't use this pointer to change the thing being pointed to

- <u>Tip</u>: 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;
}
```

constmadness.cc

`const` and Pointers

❖ The syntax with pointers is confusing:

```cpp
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 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!

```cpp
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;
}
```

Poll Everywhere

pollev.com/cse333

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

poll2.cc

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...**

```cpp
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;
}
```

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)
  - Or use `const` references (for complex struct/object instances)
- Output parameters:
  - 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
```

- 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`

- This means that if 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

```cpp
#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_
```

Class Member Definitions (`.cc` file)

Point.cc

```cpp
#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;
}
```

Class Usage (`.cc` file)

usepoint.cc

```cpp
#include <iostream>
#include <cstdlib>
#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 EXIT_SUCCESS;
}
```

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…