

C++ References, Const, Classes

CSE 333 Summer 2018

Instructor: Hal Perkins

Teaching Assistants:

Renshu Gu

William Kim

Soumya Vasisht

Administrivia

- ❖ Exercise 9 released today, due Friday

- ❖ Homework 2 due next Thursday (7/19)
 - File system crawler, indexer, and search engine
 - Note: `libhw1.a` (yours or ours) needs to be in right directory
 - Demo: use Ctrl-D to exit `searchshell`, test on directory of small self-made files
 - Valgrind takes a *long* time on the full `test_tree`. Try using `enron docs only` or other small test data directory

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;  
}
```



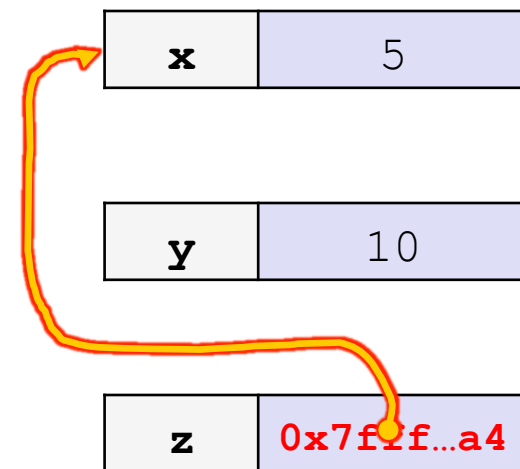
pointer.cc

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.

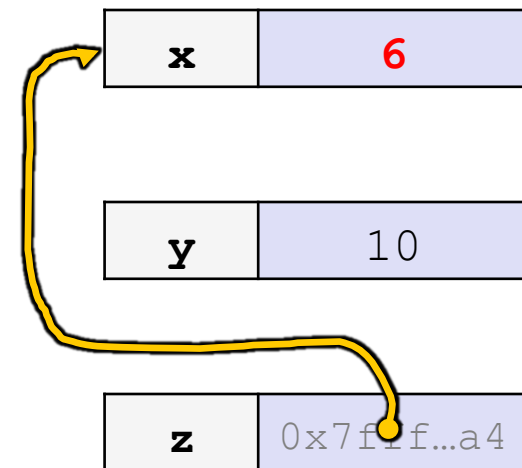
- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

```
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.

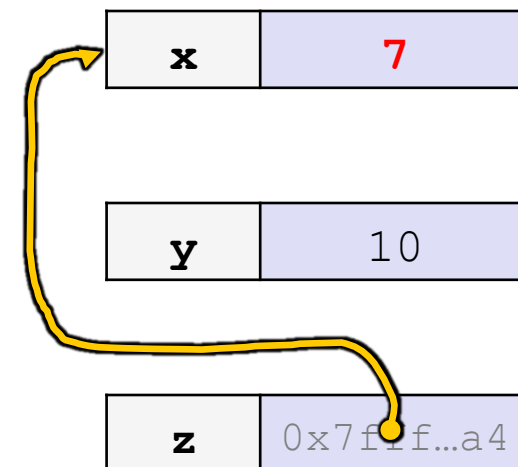
- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

```
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.

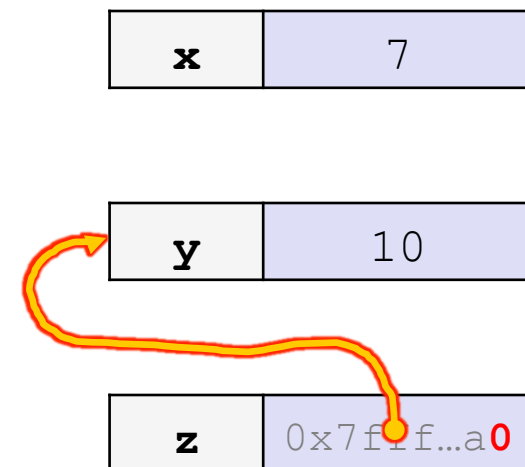
- ❖ A **pointer** is a variable containing an address
 - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
 - These work the same in C and C++

```
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;
}
```



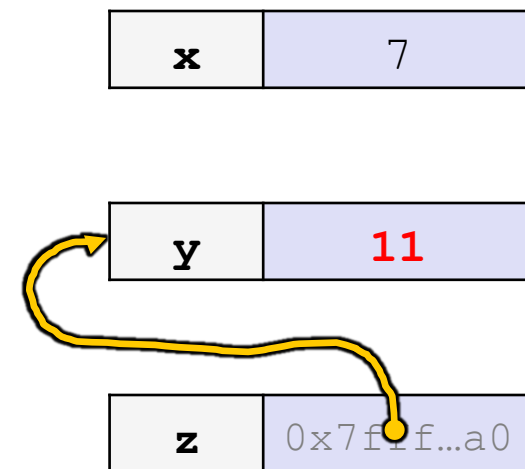
pointer.cc

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

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

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

Pass-By-Reference

Note: Arrow points to *next* instruction.

- ❖ C++ allows you to truly *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	5
-----------------	---

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

Pass-By-Reference

Note: Arrow points to *next* instruction.

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

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

(main) a (swap) x	5
------------------------------------	---

(main) b (swap) y	10
------------------------------------	----

(swap) tmp	
-------------------	--

Pass-By-Reference

Note: Arrow points to *next* instruction.

- ❖ C++ allows you to truly 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	5
(swap) x	

(main) b	10
(swap) y	

(swap) tmp	5
-------------------	----------

Pass-By-Reference

Note: Arrow points to *next* instruction.

- ❖ C++ allows you to truly 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 truly 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	10

(main) b	5
(swap) y	5

(swap) tmp	5
-------------------	---

Pass-By-Reference

Note: Arrow points to *next* instruction.

- ❖ C++ allows you to truly 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
-----------------	---

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
 - 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
 - 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++;                // 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
 - It would be BAD if you passed it a `const` variable

```
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
    bar(&b);    // OK

    return EXIT_SUCCESS;
}
```

Style Guide Tip

- ❖ Use `const` reference parameters for input values
 - Particularly for large values (no copying)
- ❖ Use pointers for output parameters
 - Input parameters first, then output parameters last

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

int main(int argc, char** argv) {
    int w = 10, h = 20, a;
    CalcArea(w, h, &a);
    return EXIT_SUCCESS;
}
```

styleguide.cc

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

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`

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_
```


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;
}
```

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;
}
```

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...
 - Should we start class with a quiz next time?

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