C++ References, Const
CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:
Dao Yi           Farrell Fileas           Lukas Joswiak
Nathan Lipiarski Renshu Gu              Travis McGaha
Yibo Cao         Yifan Bai              Yifan Xu
About how long did Exercise 7 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I prefer not to say
Administrivia

- Exercise 8 released today, due Friday
  - First C++ exercise!
  - Some parallels to ex0; compare between C/C++

- `#include <stdlib.h>` or `<cstdlib>` for EXIT_SUCCESS
Lecture Outline

❖ Intro to C++, continued
❖ C++ References
❖ const in C++
Let’s Refine It a Bit

C++’s standard library has a `std::string` class

- Include the `string` header to use it
  - Seems to be automatically included in `iostream` on CSE Linux environment (C++11) – but include it explicitly anyway if you use it
Let’s Refine It a Bit

The **using** keyword introduces a namespace (or part of) into the current region

- **using namespace std;** imports all names from `std::`
- **using std::cout;** imports only `std::cout` (used as `cout`)

```cpp
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;

int main(int argc, char **argv) {
    string hello("Hello, World!");
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```

**helloworld2.cc**
Let’s Refine It a Bit

Here we are instantiating a `std::string` object on the stack (an ordinary local variable)

- Passing the C string "Hello, World!" to its constructor method
- `hello` is deallocated (and its destructor invoked) when `main` returns: RAII!
Let’s Refine It a Bit

The C++ string library also overloads the `<<` operator

- Defines a function (not an object method) that is invoked when the LHS is `ostream` and the RHS is `std::string`

```cpp
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;

int main(int argc, char **argv) {
    string hello("Hello, World!");
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```
String Concatenation

The string class overloads the “+” operator

- Creates and returns a new string that is the concatenation of the LHS and RHS

```cpp
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

int main(int argc, char **argv) {
    string hello("Hello");
    hello = hello + ", World!";
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```
String Assignment

The string class overloads the “=” operator

- Copies the RHS and replaces the string’s contents with it
String Manipulation

This statement is complex!
- First “+” creates a string that is the concatenation of hello’s current contents and ", World!"
- Then “=” creates a copy of the concatenation to store in hello
- Without the syntactic sugar:
  - `hello.operator=(hello.operator+("", World");`
Stream Manipulators

❖ `iomanip` defines a set of stream manipulator functions
  ▪ Pass them to a stream to affect formatting

```
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

int main(int argc, char **argv) {
    cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
    cout << hex << 16 << " " << 13 << endl;
    cout << dec << 16 << " " << 13 << endl;
    return EXIT_SUCCESS;
}
```
Stream Manipulators

\[
\begin{verbatim}
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

int main(int argc, char **argv) {
    cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
    cout << hex << 16 << " " << 13 << endl;
    cout << dec << 16 << " " << 13 << endl;
    return EXIT_SUCCESS;
}
\end{verbatim}
\]

- \texttt{setw}(x) sets the width of the next field to \texttt{x}
  - Only affects the next thing sent to the output stream (\textit{i.e.} it is not persistent)
Stream Manipulators

- **hex, dec, and oct** set the numerical base for printing *integers* output to the stream
  - In effect until stream is set to another base (*i.e.* it is persistent)

```cpp
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

int main(int argc, char **argv) {
    cout << "Hi! " << setw(4) << 5 << " " << 5 << endl;
    cout << hex << 16 << " " << 13 << endl;
    cout << dec << 16 << " " << 13 << endl;
    return EXIT_SUCCESS;
}
```
C and C++

- C is (roughly) a subset of C++
  - You can still use `printf` ... but considered bad style
  - Can mix C and C++ idioms if needed to work with existing code, but avoid mixing if you can
    - Use C++(11)

```c
#include <cstdio>
#include <cstdlib>

int main(int argc, char **argv) {
    printf("Hello from C!\n");
    return EXIT_SUCCESS;
}
```

helloworld3.cc
std::cin is an object instance of class istream

- Supports the >> operator for “extraction”
  - Can be used in conditionals – (std::cin>>num) is true if successful
- Has a getline() method and methods to detect and clear errors
Lecture Outline

❖ Intro to C++, continued
❖ C++ References
❖ const in C++
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++

```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;
}
```

**Note:** Arrow points to *next* instruction.
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++

```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;
}
```

Note: Arrow points to *next* instruction.
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++

```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;
}
```

**Note:** Arrow points to *next* instruction.
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;
}
```
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++

```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;
}
```

Note: Arrow points to next instruction.
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++

```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;
}
```

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

```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;
}
```

Note: Arrow points to next instruction.
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

```c
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int &z = x;  // binds the name "z" to x
    int *p = &x
    z += 1;
    x += 1;
    z = y;
    z += 1;

    return EXIT_SUCCESS;
}
```

Note: Arrow points to next instruction.
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;
}
```

Note: Arrow points to *next* instruction.
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

```c++
int main(int argc, char **argv) {
    int x = 5, y = 10;
    int &z = x;  // binds the name "z" to x
    int *p = &x
    z += 1;    // sets z (and x) to 6
    x += 1;    // sets x (and z) to 7
    z = y;
    z += 1;
    return EXIT_SUCCESS;
}
```

Note: Arrow points to next instruction.
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

```c
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;
}
```

Note: Arrow points to next instruction.
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

```c
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

    // y = z would set y to 11
    return EXIT_SUCCESS;
}
```

Note: Arrow points to next instruction.
Pass-By-Reference

- C++ allows you to use real **pass-by-reference**
  - Client passes argument with normal “copy” syntax
  - Function uses reference parameters with normal syntax
  - ... but modifying it 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;
}
```

Note: Arrow points to next instruction.
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;
}
```

Note: Arrow points to *next* instruction.
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;
}
```

*Note: Arrow points to next instruction.*
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(b, a) binds x⇒b & y⇒a
    swap(a, b); // binds x⇒a & y⇒b
    cout << "a: " << a << " b: " << b << endl;
    return EXIT_SUCCESS;
}
```

Note: Arrow points to next instruction.
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;
}
```

Note: Arrow points to next instruction.
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;
}
```

*Note: Arrow points to next instruction.*
Lecture Outline

❖ Intro to C++, continued
❖ C++ References
❖ const in C++
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
class BrokenPrintSquare {
public:
    void BrokenPrintSquare(const int &i) {
        i = i*i; // compiler error here!
        std::cout << i << std::endl;
    }

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

    static const int kMaxSize = 100;
};
```

`brokenpassbyrefconst.cc`
**const and Pointers**

- Pointers can change data in two different contexts:
  1. Change the value of the pointer
  2. 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 = can’t change the value of the pointer
  - `const` next to data type = can’t use this pointer to change the thing being pointed to
  - **Tip:** `const` binds *leftward* (and "bounces off" if leftmost)

```
"const ALL the things!"
```

```
int *const p1
int const *p2 > equivalent
const int *p3

(const int *const p4) > equivalent
(int const *const p5)
```
**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

```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;
}
```
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. I’m not sure…
When to Use References?

❖ A stylistic choice, not mandated by the C++ language

❖ Google C++ style guide suggests:
  ▪ Input parameters have two options:
    • Pass by copy for primitive types, like `int` or small structs/objects
    • `const` references for complex structs/objects
  ▪ Output parameters:
    • `const` pointers (unchangeable pointers referencing changeable data)
  ▪ Input parameters first, then output parameters

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