References Revisited
CSE 333 Spring 2018

Instructor: Justin Hsia

Teaching Assistants:
Danny Allen  Dennis Shao  Eddie Huang
Kevin Bi      Jack Xu       Matthew Neldam
Michael Poulain Renshu Gu  Robby Marver
Waylon Huang  Wei Lin
Administrivia

- Midterm is Friday (5/4) @ 5–6 pm in GUG 220
  - No lecture on Friday!
  - 1 double-sided page of handwritten notes; reference sheet provided on exam
  - **Topics:** everything from lecture, exercises, project, etc. up through hw2 and C++ templates
  - Old exams on course website, review in section next week

- Homework 3 – spec out now, files pushed on Friday
  - Spec overview
  - Demo
Lecture Outline

- **Smart Pointers Wrap-Up**
- **References Revisited**
Smart Pointer Methods (incomplete)

- `std::unique_ptr U;`
  - `U.get()` — return to stored pointer
  - `U.release()` — release stored pointer: return value & replace with NULL
  - `U.reset(q)` — delete the stored pointer, then replace with `q`

- `std::shared_ptr S;`
  - `S.get()` — return the stored pointer
  - `S.use_count()` — return the reference count (# of `shared_ptr`)
  - `S.unique()` — equivalent to `use_count() == 1`

- `std::weak_ptr W;`
  - `W.lock()` — return a `shared_ptr` with the same stored pointer
  - `W.use_count()` — return the reference count (`shared_ptr` only)
  - `Wexpired()` — equivalent to `use_count() == 0`
GDB and Smart Pointers

- GDB demo
  - sharedvec.cc (from Lecture 16)
  - weakcycle.cc (from Lecture 16)
Lecture Outline

- Smart Pointers Wrap-Up
- References Revisited
Confusion About References

- When should they be used?
  - Particularly with parameters and return values

- When can using them cause trouble?
Peer Instruction Questions

- We’ll go through a bunch of code examples
  - Try to come up with a response on your own first
  - Then discuss with your neighbor(s) and come to a consensus
  - Vote at http://PollEv.com/justinh

A. We must NOT use a reference
B. It’s OK but discouraged to use a reference
C. It’s OK and encouraged to use a reference
D. We must use a reference
E. We’re lost...
Parameters 1

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

using namespace std;

// SHOULD WE BE USING REFERENCES FOR PARAMETERS "a" AND "b"?
// (Answer: ?)
int LeastCommonMultiple(const int &a, const int &b) {
    for (int n=1; ; n++) {
        if ((n % a == 0) && (n % b == 0))
            return n;
    }
}

int main(int argc, char **argv) {
    int x = 12, y = 14;

    int lcm = LeastCommonMultiple(x, y);
    cout << "LCM(" << x << "," << y << ") is " << lcm << endl;
    return EXIT_SUCCESS;
}
```
param1.cc

B. It’s OK but *discouraged* to use a reference

- A const reference to a small primitive type (e.g. `int`, `float`)
- We aren’t changing the argument values (`const`), so it doesn’t matter if we use a copy or not – reference is *optional*
- Correct behavior, but questionable performance benefit

---

*no major benefit to preventing copy of small pieces of data*
Parameters 2

```cpp
#include <cmath>
#include <cstdlib>
#include <iostream>
#include "ThreeDPoint.h"

// SHOULD WE BE USING REFERENCES FOR PARAMETERS "a" AND "b"?
// (Answer: ?)
double Distance(const ThreeDPoint &a, const ThreeDPoint &b) {
    double dist = pow(a.x-b.x, 2) + pow(a.y-b.y, 2) + pow(a.z-b.z, 2);
    return sqrt(dist);
}

int main(int argc, char **argv) {
    ThreeDPoint a(1, 2, 3), b(4, 5, 6);

    int dist = Distance(a, b);
    cout << "Distance(a,b) is " << dist << endl;
    return EXIT_SUCCESS;
}
```
param2.cc

C. It’s OK and encouraged to use a reference

- A const reference to a complex type (e.g. struct, object instance)
- We aren’t changing the argument values (const), so it doesn’t matter if we use a copy or not – reference is optional
- Correct behavior and likely performance benefit from not having to copy

Follow-up: Why not pass in a pointer instead?

- Can achieve all of the same behavior with a pointer as reference.
- Input parameters should be const ref
- Output parameters should be pointers
# include <cstdlib>
# include <iostream>

typedef struct Point_st {
   double x, y, z;
} Point;

// SHOULD WE BE USING A REFERENCE FOR THE RETURN VALUE?
// (Answer: ?)
Point & MakePoint(const int x, const int y, const int z) {
   Point retval = {x, y, z};
   return retval;
}

int main(int argc, char **argv) {
   Point p = MakePoint(1, 2, 3);
   std::cout << p.x << "," << p.y << "," << p.z << std::endl;
   return EXIT_SUCCESS;
}
ret1.cc

A. We must NOT use a reference

- A reference to a stack-allocated complex type
- Never return a reference (or pointer to) a local variable
  - Destructor is also called on object when returning
Copy Constructor

```cpp
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

#include <iostream>

namespace complex {

class Complex {

public:
   // Copy constructor -- should we pass a reference or not?
   // (Answer: ?)
   Complex(const Complex &copyme) {
      real_ = copyme.real_;        // real
      imag_ = copyme.imag_;        // imaginary
   }

private:
   double real_, imag_;         // real
}; // class Complex

} // namespace complex

#endif // _COMPLEX_H_
```
Complex1.h

D. We must use a reference

- A const reference to a complex type
- We aren’t changing the argument’s values so it doesn’t matter if we use a copy or not, in theory
- A copy constructor must take a reference, otherwise it would need to call itself to get a copy of the argument...
operator+

#include <iostream>

namespace complex {

class Complex {
    public:
        // Should operator+ return a reference or not?
        // (Answer: ?)
        Complex &operator+(const Complex &a) const {
            Complex tmp(0,0);
            tmp.real_ = this->real_ + a.real_;
            tmp.imag_ = this->imag_ + a.imag_;
            return tmp;
        }

    private:
        double real_, imag_;  
};  // class Complex

} // namespace complex

// Complex2.h
Complex2.h

A. We must NOT use a reference

- A reference to a stack-allocated complex type
- Never return a reference (or pointer to) a local variable
  - Destructor is also called on object when returning

Follow-up: If we fix the code, does chaining work?

Yes.
Assignment Operator

```cpp
#include <iostream>

namespace complex {

class Complex {

public:
    // Should the assignment operator return a reference?
    // (Answer: ?)
    Complex &operator=(const Complex &a) {
        if (this != &a) {
            this->real_ = a.real_;  
            this->imag_ = a.imag_;  
        }
        return *this;
    }

    private:
        double real_, imag_;  
};  // class Complex

} // namespace complex
```
D. We must use a reference

- A reference to \(*\text{this}\), the object this method was called on
- All of the “work” is done in the method body; the return value is only there for chaining (but \textit{required} for chaining to work correctly)

Follow-up: What happens in \((a = b) = c;\) if we don’t use a reference?


```cpp
# include <iostream>

namespace complex { 

class Complex {

public:

   // Should += return a reference?
   // (Answer: ?)
   Complex &operator+=(const Complex &a) {
      this->real_ += a.real_; 
      this->imag_ += a.imag_; 
      return *this;
   }

private:

   double real_, imag_; 
}; // class Complex 

} // namespace complex 
```
Complex4.h

D. **We must use a reference**

- A reference to `*this`, the object this method was called on
- All of the “work” is done in the method body; the return value is only there for chaining (but *required* for chaining to work correctly)
- You hardly see people chain `+=`, but it is allowed by the primitive data types, so we follow suit
#include <iostream>

namespace complex {

class Complex {
    public:
        double real() const { return real_; }
        double imag() const { return imag_; }

    private:
        double real_, imag_;
}; // class Complex

}; // namespace complex

// Should operator<< return a reference?
// (Answer: ?
std::ostream &operator<<(std::ostream &out,
            const complex::Complex &a) {
    out << "(" << a.real() << " + " << a.imag() << "i)";
    return out;
}

Complex5.h
Complex5.h

D. We must use a reference

- A reference to `out`, the ostream object provided as an reference argument
- The return value is only there for chaining (but *required* for chaining to work correctly)
- Copying of streams is disallowed (and doesn’t make sense)