pollev.com/cse333dylan

# About how long did Exercise 7 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++ Smart Pointers

## CSE 333 Winter 2022

**Guest Instructor:** Dylan Hartono

**Teaching Assistants:**

| | | |
|---|---|---|
| Aakash Srazali | Assaf Vayner | Brenden Page |
| Cleo Chen | Dan Constantinescu | Dylan Hartono |
| Elizabeth Haker | Jacob Christy | Julia Wang |
| Kenzie Mihardja | Kyrie Dowling | Mengqi Chen |
| Mitchell Levy | Timmy Yang | |

# Relevant Course Information

- Midterm starts Wednesday (2/9) and runs until end of Saturday (2/12)
  - **Topics:** everything from lecture, exercises, project, etc. up through hw2 and ex7
  - Written answers – short-answer questions and text file uploads
  - Gradescope quiz – can open, close, & submit as much as you want
  - Some discussion allowed if following the *Gilligan's Island Rule*

- Exercise 8 released today and due **Wednesday (2/16)** at 11am PDT
  - Practice using C++ STL containers

# Lecture Outline

- **Introducing STL Smart Pointers**
  - `ToyPtr` refresher
  - Reference Counting, `shared_ptr` (and `weak_ptr`)
  - `unique_ptr`

- Possible Errors with Smart Pointers
  - `weak_ptr` and Reference Counting Cycles
  - Smart Pointer gotcha's
  - Handling multiple Smart Pointers

# Motivations for Smart Pointers

- Automatically manage allocated memory
  - I don't have to call `delete` or `delete[]` on memory
  - Memory will deallocate when I'm not using it anymore
  - Decrease programming overhead of managing memory

- Work similarly to using a normal pointer
  - I can access a pointer using -> and *
  - I can also change the value that I am dereferencing

# Refresher: ToyPtr Class Template

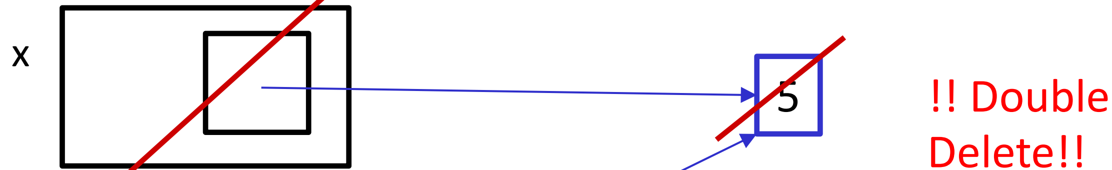
ToyPtr.h

```
#ifndef TOYPTR_H_
#define TOYPTR_H_

template <typename T>
class ToyPtr {
 public:
  ToyPtr(T* ptr) : ptr_(ptr) { }     // constructor
  ~ToyPtr() { delete ptr_; }         // destructor


  T& operator*() { return *ptr_; }  // * operator
  T* operator->() { return ptr_; }  // -> operator

 private:
  T* ptr_;                         // the pointer itself
};

#endif  // TOYPTR_H_
```

# ToyPtr Class Issue

Toy_UseToyPtr.cc

```
#include "ToyPtr.h"

// We want two pointers!
int main(int argc, char** argv) {
  ToyPtr<int> x(new int(5));
  ToyPtr<int> y(x);
  return EXIT_SUCCESS;
}
```




!! Double Delete!!

Any thoughts on designing around this?

# Smart Pointers Solutions

- Solution 1: Reference Counting
  - `shared_ptr` (and `weak_ptr`)
  - Counting the number of references (*i.e.* pointers that hold the address, not C++ references) to an object
  - Only deallocating the pointer when no other smart pointers are managing the pointer
- Solution 2: Single Ownership of Memory
  - `unique_ptr`
  - A single smart pointer will have **sole ownership** over a pointer of heap memory

# Solution 1: Reference Counting (`shared_ptr`)

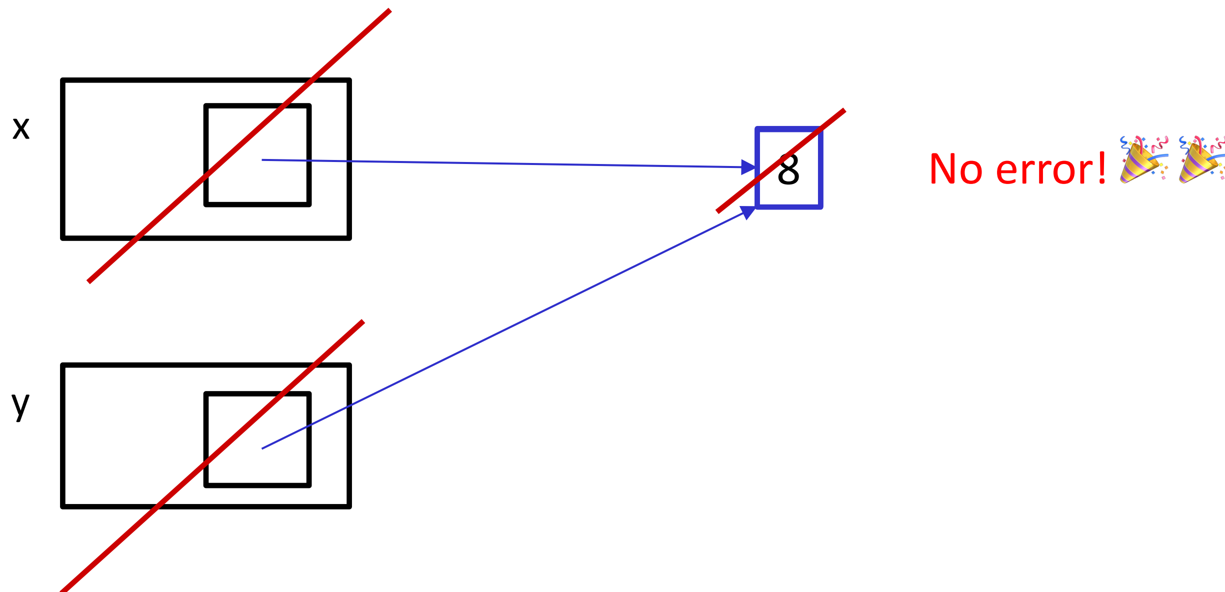
- `shared_ptr` is similar to our `ToyPtr` but implements **reference counting**
  - https://en.cppreference.com/w/cpp/memory/shared_ptr
  - It counts the number of **references** to an object
  - Managed abstractly through sharing a resource counter
    - ctors will create the counter
    - Assignment/cctors increment the counter
    - dtors decrement the counter and free

- Memory is freed when the reference count is **0**
  - All `shared_ptr`s have fallen out of scope
  - Assumes that the memory being stored is allocated **on the heap**

# Now using `shared_ptr`

Shared_Usage.cc

```
#include <memory>     // for std::shared_ptr

// We want two pointers!
int main(int argc, char** argv) {
  std::shared_ptr<int> x(new int(5));
  *x += 3;
  std::shared_ptr<int> y = x;
  return EXIT_SUCCESS;
}
```



# shared_ptrs and STL Containers

- Use shared_ptrs inside STL Containers
  - Avoid extra object copies
  - Safe to do, since copy/assign maintain a shared reference count

Shared_Vector.cc

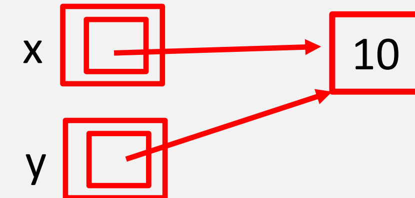
```
vector<std::shared_ptr<int> > vec;

vec.push_back(std::shared_ptr<int>(new int(9)));
vec.push_back(std::shared_ptr<int>(new int(5)));
vec.push_back(std::shared_ptr<int>(new int(7)));

int& z = *vec[1];
std::cout << "z is: " << z << std::endl;

std::shared_ptr<int> copied(vec[1]);  // works!
std::cout << "*copied: " << *copied << std::endl;

vec.pop_back(); // removes with deallocating 7!
```

# Practice with Reference Counts

- What is the expected output of this program?
- When does all memory get deallocated?

ReferenceCount_Share.cc

```
... (assume necessary includes)

int main(int argc, char** argv) {
  std::shared_ptr<int> x(new int(10));
  std::cout << x.use_count() << std::endl;

  // temporary inner scope (!)
  {
    std::shared_ptr<int> y(x);
    std::cout << y.use_count() << std::endl;
  }
  std::cout << x.use_count() << std::endl;
  std::cout << x.unique() << std::endl;

  return EXIT_SUCCESS;
}
```




Output:
1
2
1
true

# Solution 2: Unique Ownership (`unique_ptr`)

- A `unique_ptr` is the *sole owner* of a pointer to memory
  - https://en.cppreference.com/w/cpp/memory/unique_ptr
  - Similar operators to `shared_ptr` *without reference counting*
  - When the `unique_ptr` falls out of scope, it will call `delete` on the managed pointer

- Enforces uniqueness by disabling copy and assignment
  - Creates a compiler error if a `unique_ptr` is copied/assigned

- As an owner, a `unique_ptr` can choose to transfer and release ownership of a pointer

# unique_ptrs Cannot Be Copied

- std::unique_ptr has disabled its copy constructor and assignment operator
  - You cannot copy a unique_ptr, helping maintain "uniqueness" or "ownership"

Unique_Fail.cc

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::unique_ptr<int> x(new int(5));  // ctor that takes a pointer ✓

  std::unique_ptr<int> y(x);           // cctor, disabled. compiler error ✗

  std::unique_ptr<int> z;              // default ctor, holds nullptr ✓

  z = x;                               // op=, disabled. compiler error ✗

  return EXIT_SUCCESS;
}
```

# `unique_ptr`s and STL

- `unique_ptr`s *can* also be stored in STL containers

- Contradiction! STL containers make copies of stored objects and `unique_ptr`s cannot be copied

- But each element in a container is generally only going to have a sole pointer
  - Shouldn’t there be a way to do this to not have to keep track of reference count too?

# Releasing and Transferring Ownership

- As an "owner" to a pointer, `unique_ptr`s should be able to remove its ownership
  - **`release`** and **`reset`** free ownership of a `unique_ptr`

Unique_Ownership.cc

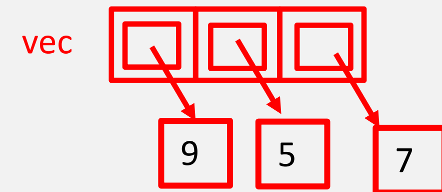
```
int main(int argc, char** argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << *x << endl;
  // Releases ownership and returns a pointer
  unique_ptr<int> y(x.release());  // x gives ownership to y
  cout << "y: " << *y << endl;
  unique_ptr<int> z(new int(10));
  // y gives ownership to z
  // z's reset() deallocates "10" and stores y's pointer
  z.reset(y.release());
  return EXIT_SUCCESS;
}
```

# `unique_ptr` and STL Example

- STL's supports transfer ownership of `unique_ptr`s using **move** semantics

Unique_Vector.cc

```
int main(int argc, char** argv) {
  std::vector<std::unique_ptr<int> > vec;

  vec.push_back(std::unique_ptr<int>(new int(9)));
  vec.push_back(std::unique_ptr<int>(new int(5)));
  vec.push_back(std::unique_ptr<int>(new int(7)));

  // z holds 5
  int z = *vec[1];
  std::cout << "z is: " << z << std::endl;

  // compiler error!
  std::unique_ptr<int> copied(vec[1]);


  return EXIT_SUCCESS;
}
```



# unique_ptr and Move Semantics

- "Move semantics" (as compared to "Copy semantics") move values from one object to another without copying
  - https://en.cppreference.com/w/cpp/language/move_constructor
  - Useful for optimizing away temporary copies
  - STL's use move semantics to change ownership of unique_ptrs

Semantics_Move.cc

```
... (includes and other examples)
int main(int argc, char** argv) {
  std::unique_ptr<string> a(new string("Hello"));

  // moves a to b
  std::unique_ptr<string> b = std::move(a);
  std::cout << "a: " <<  a << std::endl; // default ctor value
  std::cout << "b: " << *b << std::endl; // "Hello"

  return EXIT_SUCCESS;
}
```

# Choosing Between Smart Pointers

- `shared_ptr`s allow multiple pointers manage the same memory
  - Reference counting allows to deallocate when every smart pointer has stopped using it
  - Used a lot more (more purposes with shared owners)

- `unique_ptr`s help showing ownership of memory
  - The owner is responsible for calling `free/delete` when it's time to delete the resource
    - Recall in HW1 & HW2, we specifically documented who takes ownership of a resource
  - Less overhead. There's no additional resource needed for reference counting (since there is none)

# Lecture Outline

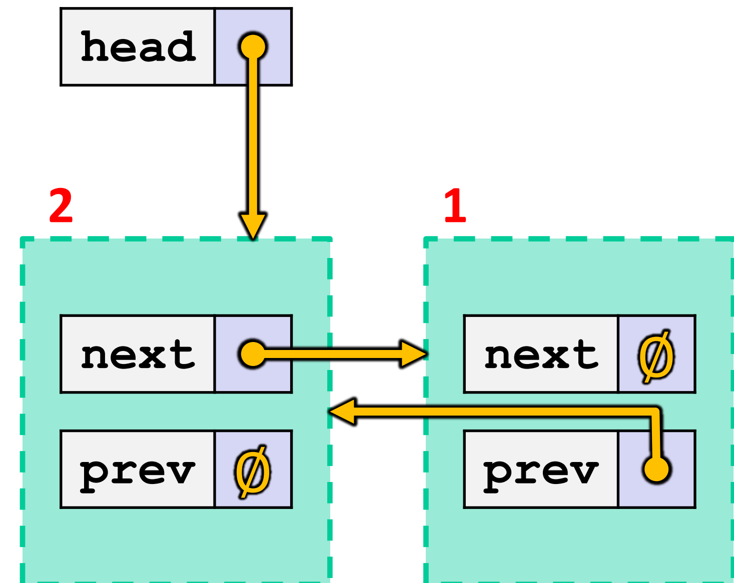
- Introducing STL Smart Pointers
  - `ToyPtr` refresher
  - Reference Counting, `shared_ptr` (and `weak_ptr`)
  - `unique_ptr`

- **Possible Errors with Smart Pointers**
  - `weak_ptr` and Reference Counting Cycles
  - Smart Pointer gotcha's
  - Handling multiple Smart Pointers

# Reference Counting: Cycle of `shared_ptr`s

Cycle_Shared.cc

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  shared_ptr<A> prev;
};

int main(int argc, char** argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```




- What happens when `main` returns?

# Solution: `weak_ptr`s

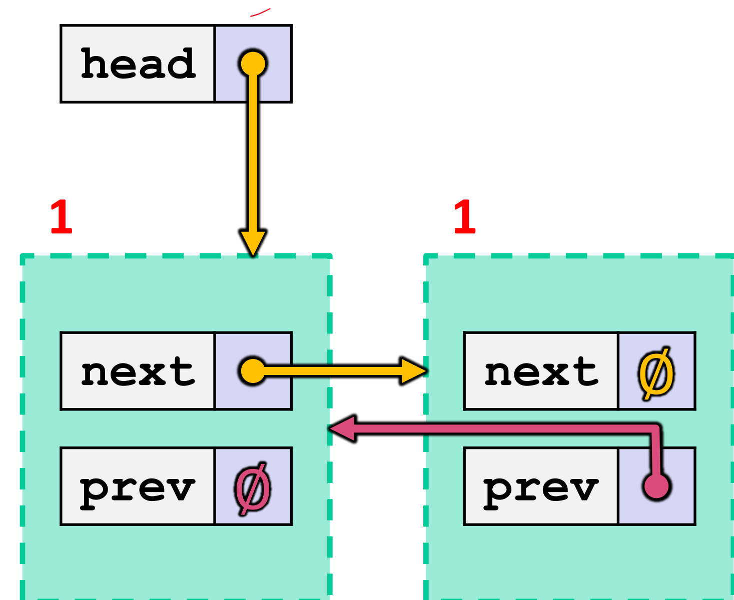
- `weak_ptr` is similar to a `shared_ptr` but *doesn't affect* the **reference count**
  - https://en.cppreference.com/w/cpp/memory/weak_ptr
  - Not really a pointer as it **cannot** be dereferenced
  - But you can use the **`lock`** function to "promote" it to an associated `shared_ptr`

- But it can be used to break our cycle problem!

# Breaking the Cycle with `weak_ptr`

Cycle_Weak.cc

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;
};

int main(int argc, char** argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```




- Now what happens when `main` returns?

# Reference Counting: Dangling weak_ptr

- weak_ptrs don't change reference count and can become "*dangling*"
  - Object referenced may have been delete'd

ReferenceCount_Weak.cc

```
... (includes and other examples)
int main(int argc, char** argv) {
  std::weak_ptr<int> w;

  {  // temporary inner scope
    std::shared_ptr<int> y(new int(10));
    w = y; // assignment operator of weak_ptr takes a shared_ptr
    std::shared_ptr<int> x = w.lock();  // "promoted" shared_ptr

    std::cout << *x << " " << w.expired() << std::endl;
  }
  std::cout << w.expired() << std::endl;
  w.lock();  // returns a nullptr

  return EXIT_SUCCESS;
}
```

# Lecture Outline

- Introducing STL Smart Pointers
  - `ToyPtr` refresher
  - Reference Counting, `shared_ptr` (and `weak_ptr`)
  - `unique_ptr`

- **Possible Errors with Smart Pointers**
  - `weak_ptr` and Reference Counting Cycles
  - Smart Pointer gotcha's
  - Handling multiple Smart Pointers

# Limitations with Smart Pointers

- Although smart pointers help with managing memory, they follow some guidelines to enforce this
  - Need to be careful with how you manage smart pointers

# Using a non-heap pointer

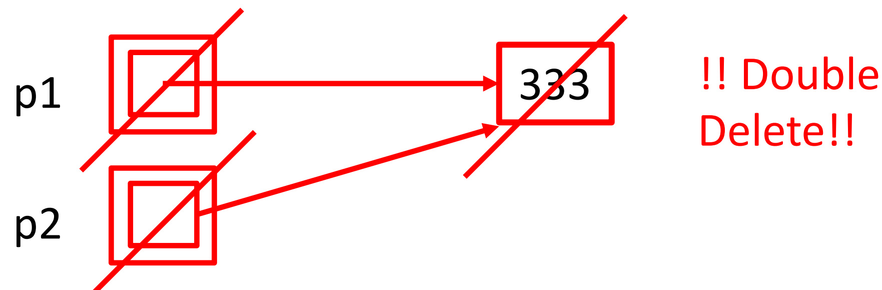
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

int main(int argc, char** argv) {
  int x = 333;

  shared_ptr<int> p1(&x);

  return EXIT_SUCCESS;
}
```

- Smart pointers can't tell if the pointer you gave points to the heap!
  - Will still call delete on the pointer when destructed.

# Re-using a raw pointer

```
#include <cstdlib>
#include <memory>

using std::unique_ptr;


int main(int argc, char** argv) {
  int* x = new int(333);

  unique_ptr<int> p1(x);

  unique_ptr<int> p2(x);

  return EXIT_SUCCESS;
}
```

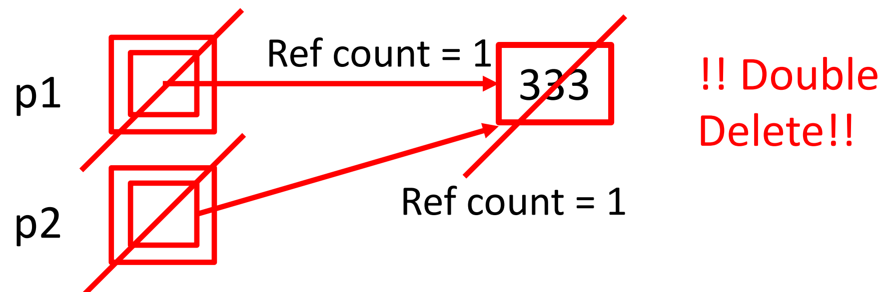
- Smart pointers can't tell if you are re-using a raw pointer.



# Re-using a raw pointer

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

int main(int argc, char** argv) {
  int* x = new int(333);

  shared_ptr<int> p1(x);  // ref count: 1

  shared_ptr<int> p2(x);  // ref count: 1

  return EXIT_SUCCESS;
}
```

- Smart pointers can't tell if you are re-using a raw pointer.



# Re-using a raw pointer: Fixed Code

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;


int main(int argc, char** argv) {
  int* x = new int(333);

  shared_ptr<int> p1(new int(333));

  shared_ptr<int> p2(p1); // ref count:

  return EXIT_SUCCESS;
}
```

(The line `int* x = new int(333);` is struck through.)

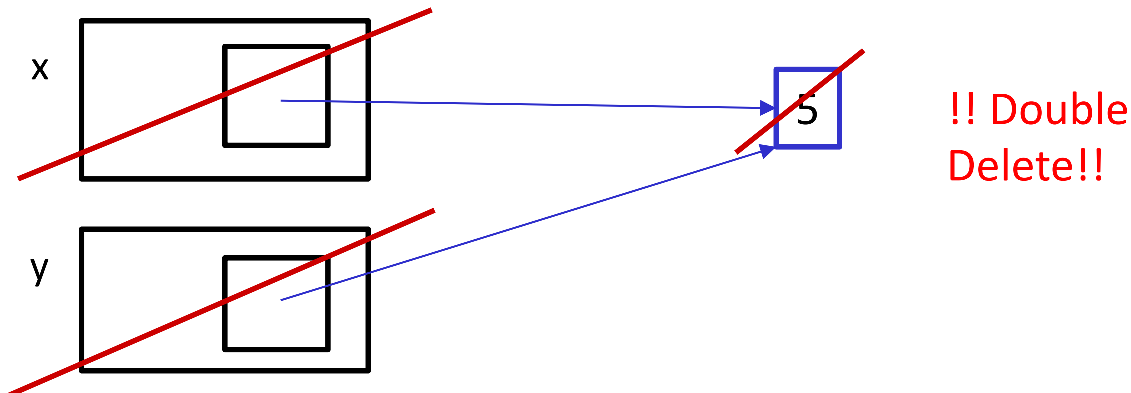
- Smart pointers can't tell if you are re-using a raw pointer.
  - Takeaway: be careful!!!!
  - Safer to use cctor
  - To be extra safe, don't have a raw pointer variable!

# Caveat: Problems from get()

- Smart pointers still have functions to return the raw pointer without losing its ownership
  - get() can circumvent smart pointer usage

UseToyPtr.cc

```
#include <memory>

// Trying to get two pointers to the same thing
int main(int argc, char** argv) {
  unique_ptr<int> x(new int(5));
  unique_ptr<int> y(x.get());
  return EXIT_SUCCESS;
}
```




!! Double Delete!!

# Summary of Smart Pointers

- A `shared_ptr` utilizes *reference counting* for multiple owners of an object in memory
  - `delete`s an object once its reference count reaches zero
- A `weak_ptr` works with a shared object but doesn't affect the reference count
  - Can't actually be dereferenced, but can check if the object still exists and can get a `shared_ptr` from the `weak_ptr` if it does
- A `unique_ptr` ***takes ownership*** of a pointer
  - Cannot be copied, but can be moved