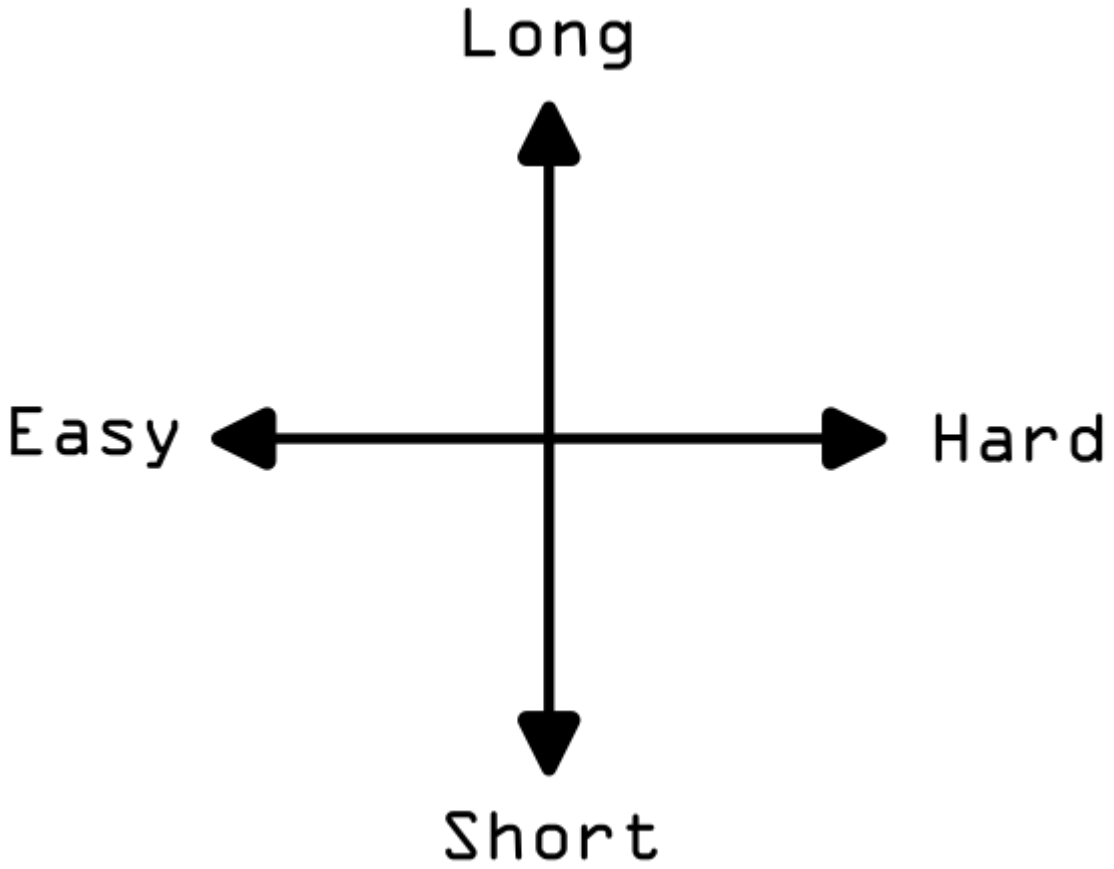
Poll Everywhere

pollev.com/naomila

How was the midterm?



CSE333 Systems Programming

Smart Pointers

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

Ad! Min! Is! Trivia?! (1/2)

- Congrats on making it through the midterm.
  - We'll have your grades out by early next week
- Exercise 7 has grown enough to be released into the wild
  - Be free, little vector class!
- Exercise 8 has stumbled into the room and looks dehydrated, in need of love
  - Counting word frequencies
  - Practice using `std::map`

Ad! Min! Is! Trivia?! (2/2)

- Homework 3 was released on Monday, due 5/21
  - Using C++ to read and write your search index to disk
  - Spec includes two overview videos to get you acquainted with the starter code

- Mid-quarter survey
  - TBD but not due on Friday

Lecture Outline

- **Introducing Smart Pointers**
- STL Smart Pointers

Motivation

- We noticed that STL was doing an enormous amount of copying
- A solution: store pointers in containers instead of objects
  - But who's responsible for deleting and when???

C++ Smart Pointers

- A smart pointer is an *object* that stores a pointer to a heap-allocated object
  - A smart pointer looks and behaves like a regular C++ pointer
    - By overloading `*`, `->`, `[]`, etc.
  - These can help you manage memory
    - The smart pointer will **delete** the pointed-to object *at the right time* (including invoking the object's destructor!)
    - When the deletion happens depends on what kind of smart pointer you use
    - With correct use of smart pointers, you no longer have to remember when to `delete` new'd memory!

A Toy Smart Pointer

- We can implement a simple one with:
  - A constructor that accepts a pointer
  - A destructor that deletes the pointer
  - Overloaded `*` and `->` operators that access the pointer

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 Example (1/2)

usetoy.cc

```
#include <iostream>
#include "ToyPtr.h"

typedef struct { int x = 1, y = 2; } Point;
std::ostream& operator<<(std::ostream& out, const Point& rhs) {
  return out << "(" << rhs.x << "," << rhs.y << ")";
}

int main(int argc, char** argv) {
  // Create a raw ("not smart") pointer
  Point* leak = new Point;

  // Create a "smart" pointer (OK, it's still pretty basic)
  ToyPtr<Point> fix_the_leak(new Point);









  return EXIT_SUCCESS;
}
```

ToyPtr Example (2/2)

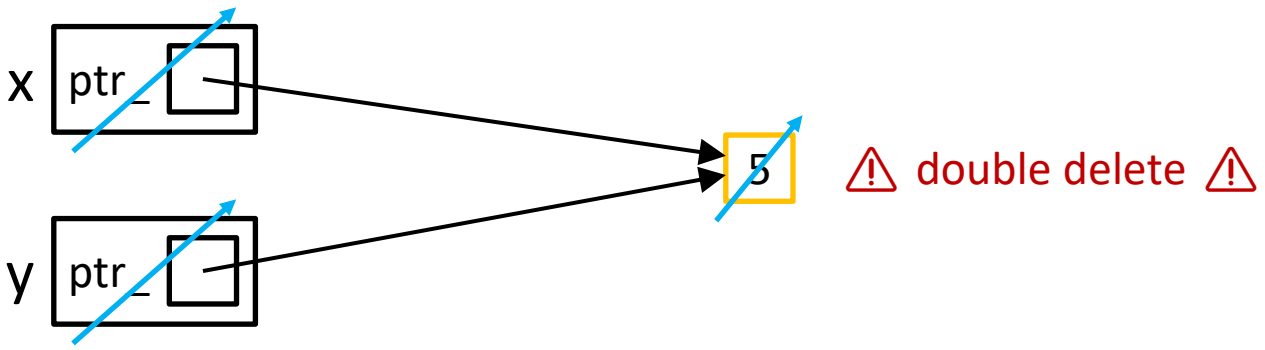
usetoy.cc

```
#include <iostream>
#include "ToyPtr.h"

typedef struct { int x = 1, y = 2; } Point;
std::ostream& operator<<(std::ostream& out, const Point& rhs) {
  return out << "(" << rhs.x << "," << rhs.y << ")";
}

int main(int argc, char** argv) {
  // Create a raw ("not smart") pointer
  Point* leak = new Point;

  // Create a "smart" pointer (OK, it's still pretty basic)
  ToyPtr<Point> fix_the_leak(new Point);

  std::cout << "          *leak: " << *leak << std::endl;
  std::cout << "        leak->x: " << leak->x << std::endl;
  std::cout << "  *fix_the_leak: " << *fix_the_leak << std::endl;
  std::cout << "fix_the_leak->x: " << fix_the_leak->x << std::endl;

  return EXIT_SUCCESS;
}
```

ToyPtr Class Issue

toyuse.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;
}
```



Brainstorm ways to design around this.

🤔 💭

In What Ways Is This a Toy?

- Can't handle:
  - Arrays
  - Copying (broke the Rule of Three!)
  - Reassignment (broke the Rule of Three!)
  - Comparison
  - … plus many other subtleties…

- Luckily, others have built non-toy smart pointers for us!
  - Thank you dearest STL 🙏

Lecture Outline

- Introducing Smart Pointers
- **STL Smart Pointers**
  - `std::shared_ptr`
  - `std::unique_ptr`

Goals for Smart Pointers

- Should automatically handle dynamically-allocated memory to decrease programming overhead of managing memory
  - Don't have to explicitly call `delete` or `delete[]`
  - Memory will deallocate when no longer in use – ties the lifetime of the data to the smart pointer object

- Should work similarly to using a normal/"raw" pointer
  - Expected/usual behavior using `->`, `*`, and `[]` operators
  - Only declaration/construction should be different

Smart Pointers Solutions

- Note: these are part of the C++ standard library (*i.e.*, have the `std::` prefix) and used via `#include <memory>`

- Option 1: **Reference Counting**
  - `shared_ptr` (and its meek sibling `weak_ptr`)
  - Track the number of references to an "owned" piece of data and only deallocate when no smart pointers are managing that data

- Option 2: **Unique Ownership of Memory**
  - `unique_ptr`
  - Disable copying (cctor, op=) to prevent sharing

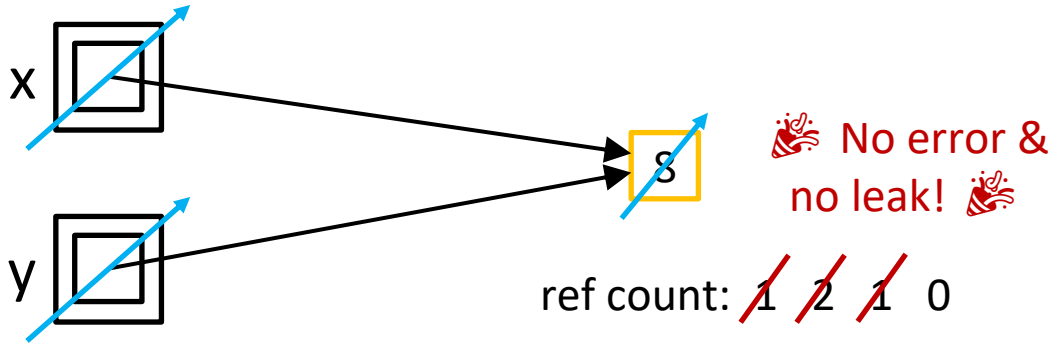
Option 1: Reference Counting

- `shared_ptr` implements **reference counting**
  - https://cplusplus.com/reference/memory/shared_ptr/
  - Counts the number of references to a piece of heap-allocated data and only deallocates it when the reference count reaches 0
    - This means that it is no longer being used and its lifetime has come to an end
  - Managed abstractly through sharing a *resource counter*:
    - Constructors will **create** the counter
    - Copy constructor and operator= will **increment** the counter
    - Destructor will **decrement** the counter

Now using shared_ptr

shareduse.cc

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

// We want two pointers!
int main(int argc, char** argv) {
  std::shared_ptr<int> x(new int(5));  // creates ref count
  *x += 3;                             // usage is the same
  std::shared_ptr<int> y(x);           // increments ref count
  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
    - Copying increments ref count, then original is destructed

sharedvec.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 smart ptr & deallocates 7!
```

Poll Everywhere

pollev.com/naomila

What is the expected output of this program?

- `use_count()` – returns reference count
- `unique()` – returns ref count == 1 (`bool`)

sharedrefcount.cc

```
... // the necessary includes are here

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;
}
```

Option 2: Unique Ownership

- A `unique_ptr` is the *sole owner* of a pointer to memory
  - https://cplusplus.com/reference/memory/unique_ptr/
  - Enforces uniqueness by disabling copy and assignment
    (compiler error if these methods are used)
    - Will therefore *always* call `delete` on the managed pointer when destructed
  - As the sole owner, a `unique_ptr` can choose to *transfer* or *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"

uniquefail.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));  // 1-arg ctor (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…

- Recall: *why* do container operations/methods create extra copies?
  - Generally to **move** things around in memory/the data structure
  - The end result is still one copy of each element – this doesn't break the sole ownership notion!

Passing Ownership

- As the "owner" of a pointer, `unique_ptr`s should be able to remove or transfer its ownership
  - **`release`**`()` and **`reset`**`()` free ownership

uniquepass.cc

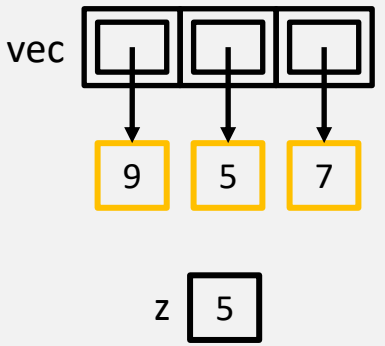
```
int main(int argc, char** argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << *x << endl;

  // Releases ownership and returns a raw 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_ptrs using **move** semantics

uniquevec.cc

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

  // moves between smart ptrs (temp to vec & capacity changes)
  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 - can read out value without changing ownership
  int z = *vec[1];
  std::cout << "z is: " << z << std::endl;

  // attempted copy semantics - 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://cplusplus.com/doc/tutorial/classes2/#move
  - Useful for optimizing away temporary copies
  - STL's use move semantics to transfer ownership of unique_ptrs instead of copying

uniquemove.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);
  // a is now nullptr (default ctor of unique_ptr)
  std::cout << "b: " << *b << std::endl; // "Hello"

  return EXIT_SUCCESS;
}
```

Choosing Between Smart Pointers

- `unique_ptr`s make ownership very clear
  - Generally the default choice due to reduced complexity – the owner is responsible for cleaning up the resource
    - Example: would make sense in HW1 & HW2, where we specifically documented who takes ownership of a resource
  - Less overhead: small and efficient

- `shared_ptr`s allow for multiple simultaneous owners
  - Reference counting allows for "smarter" deallocation but consumes more space and logic and is trickier to get right
  - Common when using more "well-connected" data structures