C++ Smart Pointers, Review
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

pollev.com/cse333

About how long did Exercise 12a take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I didn't finish / I prefer not to say

Administrivia

- HW 3 out, due Thursday 11/14
  - Save some time: read the spec!
- Midterm: Review today and in section

Lecture Outline

- **HW3 Demo**
- STL Smart Pointers
  - `unique_ptr` (finish)
  - Reference Counting and `shared_ptr` vs `weak_ptr`
- Midterm review

Lecture Outline

- HW3 Demo
- **STL Smart Pointers**
  - **`unique_ptr` (finish)**
  - Reference Counting and `shared_ptr` vs `weak_ptr`
- Midterm review

Review: `unique_ptr`

- A `unique_ptr` is the *sole owner* of its pointee
  - It will call `delete` on the pointee when it falls out of scope

- Guarantees uniqueness by disabling copy and assignment
  - But *can* be stored in STL containers thanks to move semantics and a move constructor

unique_ptr and “<”

- A unique_ptr implements some comparison operators, including operator<
  - However, it doesn't invoke operator< on the pointed-to objects
    - Instead, it just promises a stable, strict ordering (probably based on the pointer address, not the pointed-to-value)
  - So to use sort() on vectors, you want to provide it with a comparison function

unique_ptr and STL Sorting

uniquevecsort.cc

```
using namespace std;
bool sortfunction(const unique_ptr<int> &x,
                  const unique_ptr<int> &y) { return *x < *y; }
void printfunction(unique_ptr<int> &x) { cout << *x << endl; }

int main(int argc, char **argv) {
  vector<unique_ptr<int> > vec;
  vec.push_back(unique_ptr<int>(new int(9)));
  vec.push_back(unique_ptr<int>(new int(5)));
  vec.push_back(unique_ptr<int>(new int(7)));

  // buggy: sorts based on the values of the ptrs
  sort(vec.begin(), vec.end());
  cout << "Sorted:" << endl;
  for_each(vec.begin(), vec.end(), &printfunction);

  // better: sorts based on the pointed-to values
  sort(vec.begin(), vec.end(), &sortfunction);
  cout << "Sorted:" << endl;
  for_each(vec.begin(), vec.end(), &printfunction);

  return EXIT_SUCCESS;
}
```

unique_ptr and Arrays

- unique_ptr can store arrays as well
  - Will call delete[] on destruction

unique5.cc

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

using namespace std;

int main(int argc, char **argv) {
  unique_ptr<int[]> x(new int[5]);

  x[0] = 1;
  x[2] = 2;

  return EXIT_SUCCESS;
}
```

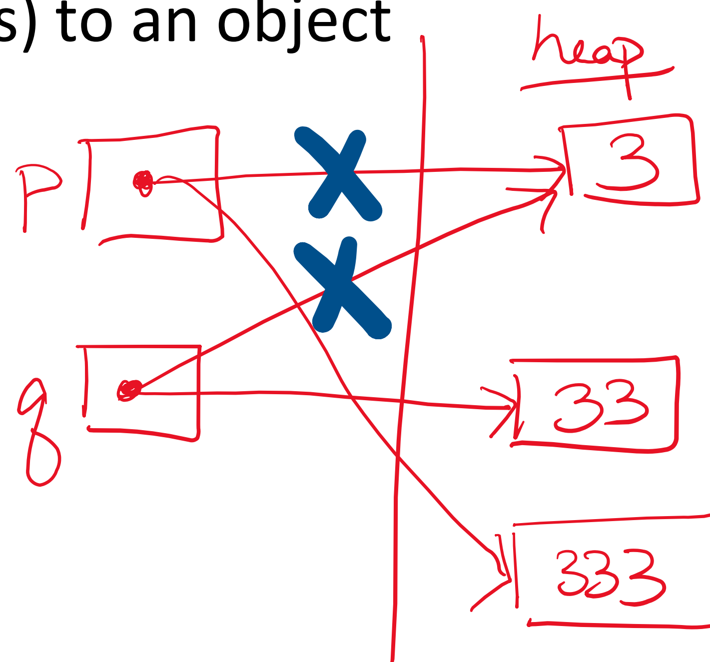
Lecture Outline

- HW3 Demo
- **STL Smart Pointers**
  - `unique_ptr` (finish)
  - **Reference Counting and `shared_ptr` vs `weak_ptr`**
- Midterm review

Reference Counting

- Reference counting is a technique for managing resources by counting and storing the number of references (*i.e.* pointers that hold the address) to an object

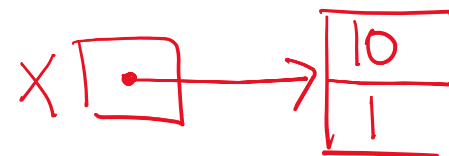
```
int *p = new int(3);
int *q = p;
q = new int(33);
p = new int(333);
```



`std::shared_ptr`

- `shared_ptr` is similar to `unique_ptr` but we allow shared objects to have multiple owners
  - The copy/assign operators are not disabled and *increment* or *decrement* reference counts as needed
    - After a copy/assign, the two `shared_ptr` objects point to the same pointed-to object and the (shared) reference count is 2
  - When a `shared_ptr` is destroyed, the reference count is *decremented*
    - When the reference count hits 0, we `delete` the pointed-to object!

shared_ptr Example



sharedexample.cc

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr

int main(int argc, char **argv) {
  std::shared_ptr<int> x(new int(10));  // ref count: 1

  // temporary inner scope (!)
  {
    std::shared_ptr<int> y = x;         // ref count: 2
    std::cout << *y << std::endl;
  }

  std::cout << *x << std::endl;         // ref count: 1

  return EXIT_SUCCESS;
}                                       // ref count: 0
```

shared_ptrs and STL Containers

- Even simpler than unique_ptrs
  - Safe to store shared_ptrs in containers, since copy/assign maintain a shared reference count

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;

std::shared_ptr<int> moved = std::move(vec[1]);  // works!
std::cout << "*moved: " << *moved << std::endl;
std::cout << "vec[1].get(): " << vec[1].get() << std::endl;
```

Cycle of shared_ptrs

strongcycle.cc

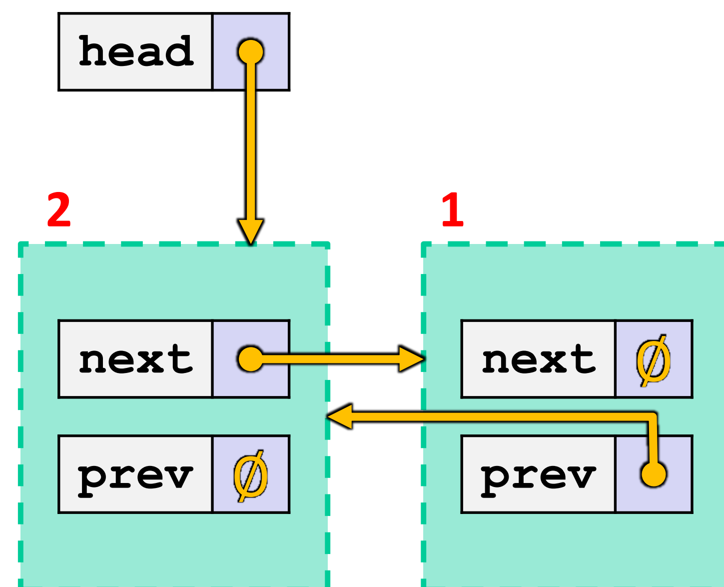
```
#include <cstdlib>
#include <memory>

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 we delete head?

std::weak_ptr

- `weak_ptr` is similar to a `shared_ptr` but doesn't affect the reference count
  - Can *only* "point to" an object that is managed by a `shared_ptr`
  - Not *really* a pointer – can't actually dereference unless you "get" its associated `shared_ptr`
  - Because it doesn't influence the reference count, `weak_ptr`s can become "*dangling*"
    - Object referenced may have been `delete`'d
    - But you can check to see if the object still exists

- Can be used to break our cycle problem!

Breaking the Cycle with `weak_ptr`

weakcycle.cc

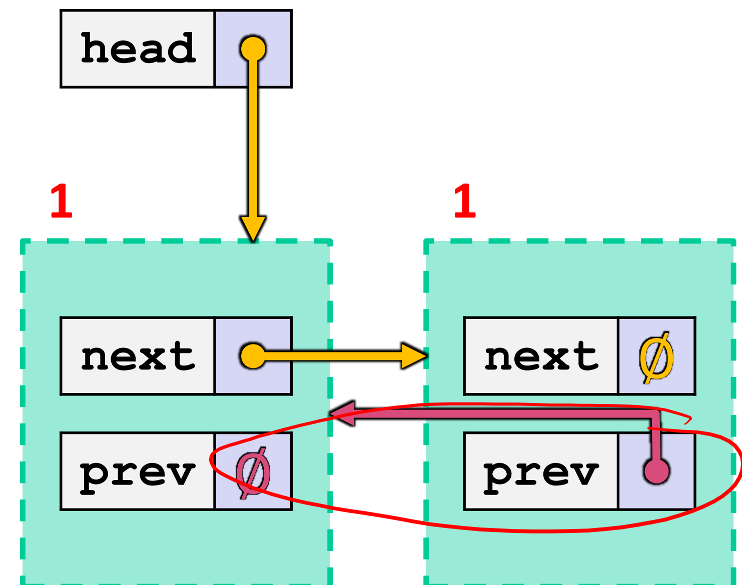
```
#include <cstdlib>
#include <memory>

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 we `delete head`?

Using a weak_ptr

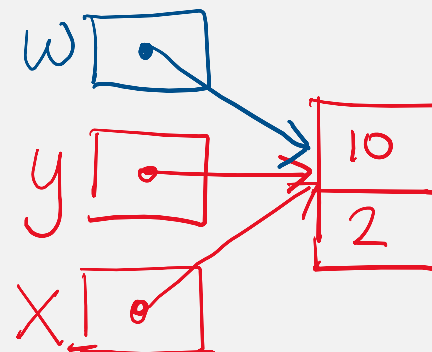
usingweak.cc

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr, std::weak_ptr

int main(int argc, char **argv) {
  std::weak_ptr<int> w;

  { // temporary inner scope
    std::shared_ptr<int> x;
    { // temporary inner-inner scope
      std::shared_ptr<int> y(new int(10));
      w = y;
      x = w.lock(); // returns "promoted" shared_ptr
      std::cout << *x << std::endl;
    }
    std::cout << *x << std::endl;
  }
  std::shared_ptr<int> a = w.lock();
  std::cout << a << std::endl;

  return EXIT_SUCCESS;
}
```



Summary

- A `unique_ptr` ***takes ownership*** of a pointer
  - Cannot be copied, but can be moved
  - **`get`**`()` returns a copy of the pointer, but is dangerous to use; better to use **`release`**`()` instead
  - **`reset`**`()` `delete`s old pointer value and stores a new one
- A `shared_ptr` allows shared objects to have multiple owners by doing *reference counting*
  - `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

Some Important Smart Pointer Methods

- `std::unique_ptr U;`
  - `U.get()`
  - `U.release()`
  - `U.reset(q)`
- `std::shared_ptr S;`
  - `S.get()`
  - `S.use_count()`
  - `S.unique()`
- `std::weak_ptr W;`
  - `W.lock()`
  - `W.use_count()`
  - `W.expired()`

Lecture Outline

- HW3 Demo
- STL Smart Pointers
  - `unique_ptr` (finish)
  - Reference Counting and `shared_ptr` vs `weak_ptr`
- **Midterm review**