C++ Smart Pointers
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

- Exercise 12a released today, due Wednesday
  - Practice using `map`

- 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
Lecture Outline

- Smart Pointers
  - `std::unique_ptr`
  - Reference counting
  - `std::shared_ptr` and `std::weak_ptr`
std::unique_ptr

- A `unique_ptr` takes ownership of a pointer
  - Part of C++'s standard library (C++11)
  - Its destructor invokes `delete` on the owned pointer
    - Invoked when `unique_ptr` object is `delete`'d or falls out of scope
#include <iostream> // for std::cout, std::endl
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

void Leaky() {
    int *x = new int(5);  // heap-allocated
    (*x)++;
    std::cout << *x << std::endl;
}  // never used delete, therefore leak

void NotLeaky() {
    std::unique_ptr<int> x(new int(5));  // wrapped, heap-allocated
    (*x)++;
    std::cout << *x << std::endl;
}  // never used delete, but no leak

int main(int argc, char **argv) {
    Leaky();
    NotLeaky();
    return EXIT_SUCCESS;
}
Why are `unique_ptrs` useful?

- If you have many potential exits out of a function, it's easy to forget to call `delete` on all of them
  - `unique_ptr` will `delete` its pointer when it falls out of scope
  - Thus, a `unique_ptr` also helps with exception safety

```cpp
void NotLeaky() {
    std::unique_ptr<int> x(new int(5));
    ...
    // lots of code, including several returns
    // lots of code, including potential exception throws
    ...
}
```
unique_ptr Operations

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

using namespace std;
typedef struct { int a, b; } IntPair;

int main(int argc, char **argv) {
    unique_ptr<int> x(new int(5));
    int *ptr = x.get(); // Return a pointer to pointed-to object
    int val = *x; // Return the value of pointed-to object
    // Access a field or function of a pointed-to object
    unique_ptr<IntPair> ip(new IntPair);
    ip->a = 100;
    // Deallocation current pointed-to object and store new pointer
    x.reset(new int(1));

    ptr = x.release(); // Release responsibility for freeing
    delete ptr;
    return EXIT_SUCCESS;
}
```
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”

```cpp
#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 arg ✓
    std::unique_ptr<int> y(x); // ctor, disabled-compiler error ✗
    std::unique_ptr<int> z; // default ctor, holds NULL ✓
    z = x; // op=, disabled-compiler error ✗

    return EXIT_SUCCESS;
}
```
Transferring Ownership

- Use `reset()` and `release()` to transfer ownership
  - `release` returns the pointer, sets wrapper’s pointer to NULL
  - `reset delete's` the current pointer and stores a new one

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

  unique_ptr<int> y(x.release()); // x abdicates ownership to y
  cout << "x: " << x.get() << endl; // NULL
  cout << "y: " << y.get() << endl; // heap addr

  unique_ptr<int> z(new int(10));
  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z.reset(y.release());

  return EXIT_SUCCESS;
}
```
unique_ptr and STL

- **unique_ptr**s can be stored in STL containers
  - Wait, what? STL containers like to make lots of copies of stored objects and **unique_ptr**s cannot be copied...

- Move semantics to the rescue!
  - When supported, STL containers will *move* rather than *copy*
    - **unique_ptr**s support move semantics
Aside: Copy Semantics

- Assigning values typically means making a copy
  - Sometimes this is what you want
    - e.g. assigning a string to another makes a copy of its value
  - Sometimes this is wasteful
    - e.g. assigning a returned string goes through a temporary copy

```cpp
std::string ReturnFoo(void) {
    std::string x("foo");
    return x; // this return might copy
}

int main(int argc, char **argv) {
    std::string a("hello");
    std::string b(a); // copy a into b
    b = ReturnFoo(); // copy return value into b

    return EXIT_SUCCESS;
}
```
Aside: Move Semantics (C++11)

- “Move semantics” move values from one object to another without copying (“stealing”)
  - Useful for optimizing away temporary copies
  - This is a complex topic, involving “rvalue references”
    - Mostly beyond the scope of 333 this quarter

```cpp
std::string ReturnFoo(void) {
    std::string x("foo");
    // this return might copy
    return x;
}

int main(int argc, char **argv) {
    std::string a("hello");
    // moves a to b
    std::string b = std::move(a);
    std::cout << "a: " << a << std::endl;
    std::cout << "b: " << b << std::endl;

    // moves the returned value into b
    b = std::move(ReturnFoo());
    std::cout << "b: " << b << std::endl;
    return EXIT_SUCCESS;
}
```
Transferring Ownership via Move

- **unique_ptr** supports move semantics
  - Can “move” ownership from one **unique_ptr** to another
    - Behavior is equivalent to the “release-and-reset” combination

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

  unique_ptr<int> y = std::move(x); // x abdicates ownership to y
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));
  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z = std::move(y);

  return EXIT_SUCCESS;
}
```
## unique_ptr and STL Example

```cpp
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 gets the value 5
    int z = *vec[1];
    std::cout << "z is: " << z << std::endl; // 5

    // compiler error - no copy constructor
    std::unique_ptr<int> copied = vec[1];

    // works, but vec[1] now holds NULL
    std::unique_ptr<int> moved = std::move(vec[1]);
    std::cout << "*moved: " << *moved << std::endl; // 5
    std::cout << "vec[1].get(): " << vec[1].get() << std::endl; // NULL (0)

    return EXIT_SUCCESS;
}
```
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

```
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 maps

- Similarly, you can use `unique_ptr`s as keys in a `map`
  - Reminder: a `map` internally stores keys in sorted order
    - Iterating through the `map` iterates through the keys in order
  - By default, “<” is used to enforce ordering
    - You must specify a comparator when constructing the `map` to get a meaningful sorted order using “<” of `unique_ptr`s

- Compare (the 3\textsuperscript{rd} template) parameter:
  - “A binary predicate that takes two element keys as arguments and returns a `bool`. This can be a `function` pointer or a `function` object.”
    - `bool fptr(T1& lhs, T1& rhs);` OR member function
      `bool operator()(const T1& lhs, const T1& rhs);`
unique_ptr and map Example

```cpp
struct MapComp {
    bool operator()(const unique_ptr<int> &lhs, const unique_ptr<int> &rhs) const { return *lhs < *rhs; }
}; // function object

int main(int argc, char **argv) {
    map<unique_ptr<int>,int,MapComp> a_map; // Create the map

    unique_ptr<int> a(new int(5)); // unique_ptr for key
    unique_ptr<int> b(new int(9));
    unique_ptr<int> c(new int(7));

    a_map[std::move(a)] = 25; // move semantics to get ownership
    a_map[std::move(b)] = 81; // of unique_ptrs into the map.
    a_map[std::move(c)] = 49; // a, b, c hold NULL after this.

    map<unique_ptr<int>,int>::iterator it;
    for (it = a_map.begin(); it != a_map.end(); it++) {
        std::cout << "key: " << *(it->first);
        std::cout << " value: " << it->second << std::endl;
    }
    return EXIT_SUCCESS;
}
```

uniquemap.cc
unique_ptr and Arrays

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

```cpp
#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;
}
```

unique5.cc
Reference Counting

- Reference counting is a technique of storing the number of references (pointers that hold the address) to an object.

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

Diagram of reference counting:

- singly-linked list:
- delete
- val
- next

- mem leak! (should delete)
**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* a reference count
    - 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

```cpp
#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 (delete!)
```

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

```cpp
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; // 5
std::cout << "vec[1].get(): " << vec[1].get() << std::endl; // NULL
```
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?

memory leak! nodes never reach ref count of zero.
std::weak_ptr

- weak_ptr is just like 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_ptrs can become “dangling”
      - Object referenced may have been delete’d

- Can be used to break our cycle problem!
Breaking the Cycle with `weak_ptr`

Now what happens when we delete `head`?

memory is cleaned up!
Using a `weak_ptr`

```cpp
#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;    // w
    { // temporary inner scope
        std::shared_ptr<int> x; // x
        { // temporary inner-inner scope
            std::shared_ptr<int> y(new int(10));
            w = y;  // w
            x = w.lock(); // returns "promoted" shared_ptr
            std::cout << *x << std::endl; // 10
        }
        std::cout << *x << std::endl; // 10
    }
    std::shared_ptr<int> a = w.lock();
    std::cout << a << std::endl;    // 0
    return EXIT_SUCCESS;
}
```
Summary

- A `unique_ptr` **takes ownership** of a pointer
  - Cannot be copied, but can be moved
  - `get()` returns the pointer, but is dangerous to use; better to use `release()` instead
  - `reset()` deletes old pointer value and stores a new one

- A `shared_ptr` allows shared objects to have multiple owners by doing **reference counting**
  - `delete` 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