C++ Smart Pointers
CSE 333 Fall 2023

Instructor: Chris Thachuk

Teaching Assistants:
Ann Baturytski  Humza Lala
Yuquan Deng    Alan Li
Noa Ferman     Leanna Mi Nguyen
James Froelich Chanh Truong
Hannah Jiang   Jennifer Xu
Yegor Kuznetsov
Relevant Course Information

- Exercise 8 due tonight by 10pm (11/6)

- Exercise 9 released tomorrow and due next Wed (11/15)
  - Practice using C++ smart pointers (this lecture), inheritance (next lecture)

- HW 3 due Nov. 23
  - Don’t delay!!

- No lecture this Friday (11/10; Veterans Day)
Lecture Outline

✓ Introducing STL Smart Pointers
  ▪ std::shared_ptr
  ▪ std::unique_ptr

✓ Smart Pointer Limitations
  ▪ std::weak_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
Example: ToyPtr Class Template

```cpp
#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.h
ToyPtr Class Issue

```cpp
#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. 😐🤔
Smart Pointers Solutions

- **Option 1: Reference Counting**
  - `shared_ptr` (and `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
  - 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`

```cpp
#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;
}
```

🎉 No error & no leak!

ref count: 1 2 1 0
shared_ptr and STL Containers

- Use shared_ptr 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

```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;
vec.pop_back();  // removes smart ptr & deallocates 7!
```
Practice with Reference Counts

- What is the expected output of this program?
  - `use_count()` – returns reference count
  - `unique()` – returns ref count == 1 (bool)

```cpp
... // 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/](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_ptr's 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));  // 1-arg ctor (pointer)  
  std::unique_ptr<int> y(x);          // ctor 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

```cpp
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

```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 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;
}
```

See full lecture code of uniquevec.cc!
**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](https://cplusplus.com/doc/tutorial/classes2/#move)
  - Useful for optimizing away temporary copies
  - STL’s use move semantics to transfer ownership of **unique_ptr**s instead of copying

```cpp
... (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;
}
```
Aside: Smart Pointers and Arrays

- Smart pointers can store arrays as well and will call `delete[]` on destruction

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

using std::unique_ptr;

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

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

    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
Lecture Outline

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

- Smart Pointer Limitations
  - `std::weak_ptr`
Limitations with Smart Pointers

- Smart pointers are only as “smart” as the behaviors that have been built into their class methods and non-member functions!

- Limitations we will look at now:
  - Can’t tell if pointer is to the heap or not
  - Circumventing ownership rules
  - Still possible to leak memory!
  - Sorting smart pointers [Bonus slides]
Using a Non-Heap Pointer

- Smart pointers will still call `delete` when destructed

```cpp
#include <cstdlib>
#include <memory>
using std::shared_ptr;

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

    shared_ptr<int> p1(&x);

    return EXIT_SUCCESS;
}
```
Re-using a Raw Pointer (**unique_ptr**)

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

```cpp
#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;
}
```
Re-using a Raw Pointer (\textit{\texttt{shared\_ptr}})

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

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

using std::shared_ptr;

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

    shared_ptr<int> p1(x);
    shared_ptr<int> p2(x);

    return EXIT_SUCCESS;
}
```

⚠️ \texttt{double delete} ⚠️
Solution: Don’t Use Raw Pointer Variables

- Smart pointers replace your raw pointers; passing `new` and then using the copy constructor is safer:

```cpp
#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);
    return EXIT_SUCCESS;
}
```
Caution Using `get()`

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

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

// Same as re-using a raw pointer
int main(int argc, char** argv) {
    unique_ptr<int> p1(new int(5));
    unique_ptr<int> p2(p1.get());
    return EXIT_SUCCESS;
}
```
Cycle of `shared_ptr`

- What happens when `main` returns?

```cpp
define <cstdlib>
define <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;
}
```
Solution: `weak_ptr`

- `weak_ptr` is similar to a `shared_ptr` but **doesn’t affect** the reference count
  - [https://cplusplus.com/reference/memory/weak_ptr/](https://cplusplus.com/reference/memory/weak_ptr/)
  - Not really a pointer as it **cannot be dereferenced** (!) – would break our notion of shared ownership
    - To deference, you first use the `lock` method to get an associated `shared_ptr`
Breaking the Cycle with `weak_ptr`

- Now what happens when `main` returns?

```cpp
#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;
}
```
Dangling `weak_ptrs`

- `weak_ptrs` don’t change reference count and can become “dangling”
  - Data referenced may have been `delete`’d

```cpp
... (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;
}
```
Summary of Smart Pointers

- A `shared_ptr` utilizes *reference counting* for multiple owners of an object in memory
  - `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, 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
Some Important Smart Pointer Methods

Visit http://www.cplusplus.com/ for more information on these!

- **std::unique_ptr**\(<T>\) `U;`
  - `U.get()` Returns the raw pointer `U` is managing
  - `U.release()` `U` stops managing its raw pointer and returns the raw pointer
  - `U.reset(q)` `U` cleans up its raw pointer and takes ownership of `q`

- **std::shared_ptr**\(<T>\) `S;`
  - `S.get()` Returns the raw pointer `S` is managing
  - `S.use_count()` Returns the reference count
  - `S.unique()` Returns true iff `S.use_count()` == 1

- **std::weak_ptr**\(<T>\) `W;`
  - `W.lock()` Constructs a shared pointer based off of `W` and returns it
  - `W.use_count()` Returns the reference count
  - `Wexpired()` Returns true iff `W` is expired (\(W.use\_count() == 0\))
BONUS SLIDES

Some details about sorting the owned data within a container of smart pointers.

These slides expand on material covered today but won’t be needed for CSE333; however, they are relevant for general C++ smart pointer usage in STL containers.
Smart Pointers and “<”

- Smart pointers implement 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)

- To use the `sort()` algorithm on a container like `vector`, you need to provide a comparison function

- To use a smart pointer in a sorted container like `map`, you need to provide a comparison function when you `declare` the container
unique_ptr and STL Sorting

```cpp
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 3rd 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; }
};

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;
}
```