

C++ Smart Pointers

CSE 333 Autumn 2021

Guest Instructor: Cosmo Wang

Teaching Assistants:

Arpad (John) Depaszthory

Ian Hsiao

Logan Gnanapragasam

Mengqi (Frank) Chen

Angela Xu

Khang Vinh Phan

Maruchi Kim

Cosmo Wang

Administrivia

- ❖ Mid-quarter survey now on canvas
 - Due Thursday @ 11:59pm
 - Helps the teaching team a lot
 - Doesn't count towards grade
 - Can waive one of ex1 – ex10 if completed
- ❖ Exercise 11 due next Monday @ 10am (11/8)
- ❖ Midterm runs all of the coming Friday (11/4)
 - Lecture cancelled on Friday so can use that time as well

Lecture Outline

❖ STL Smart Pointers

- `ToyPtr` refresher
- Reference Counting, `shared_ptr` and `weak_ptr`
- `unique_ptr`

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
};

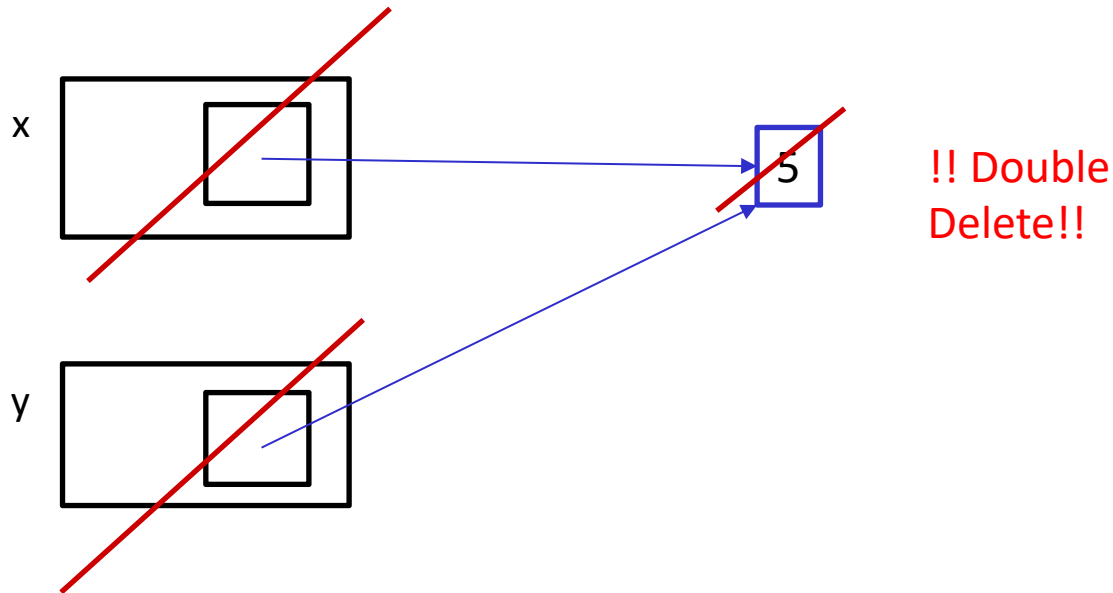
#endif // _TOYPTR_H_
```

Refresher: ToyPtr Class Template

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;
}
```

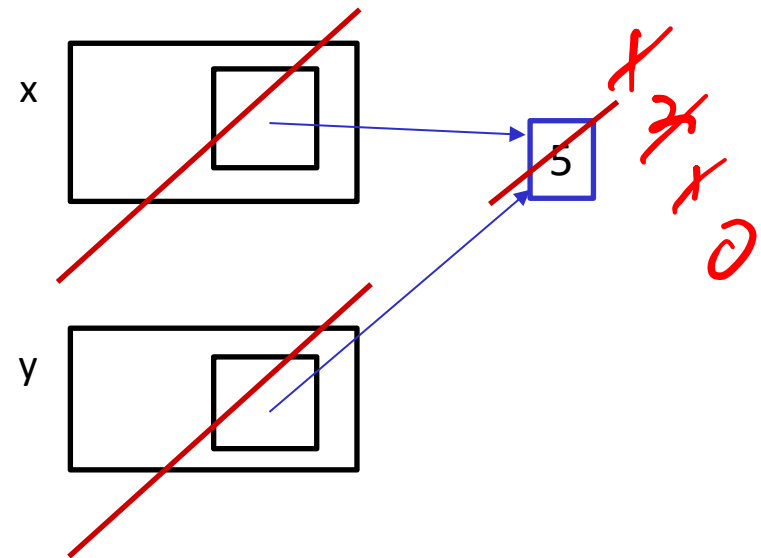


Solution: Reference Counting

- ❖ **Reference counting** is a technique for managing resources by counting and storing the number of references (i.e. pointers that hold the address, not C++ references) to an object

```
#include "../ToyPtr.h"

// Assume we have implemented
// reference counting for ToyPtr!
int main(int argc, char** argv) {
    ToyPtr<int> x(new int(5));
    ToyPtr<int> y(x);
    return EXIT_SUCCESS;
}
```



std::shared_ptr

- ❖ `shared_ptr` is similar to our `ToyPtr` but implements reference counting
 - Maintain a reference count for a managed data within the shared pointer class
 - 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

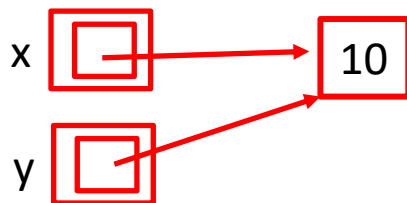
```
#include <cstdlib> // for EXIT_SUCCESS
#include <iostream> // for std::cout, std::endl
#include <memory> // for std::shared_ptr

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;

    return EXIT_SUCCESS;
}
```

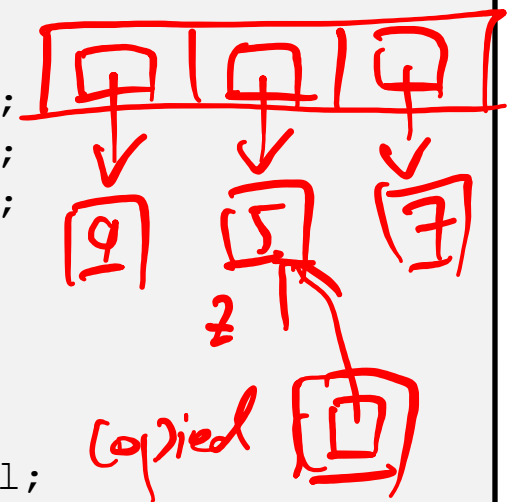


shared_ptrs and STL Containers

- ❖ Use `shared_ptr` inside STL Containers
 - Avoid extra object copies
 - Safe to do, 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;
```



Cycle of shared_ptrs

strongcycle.cc

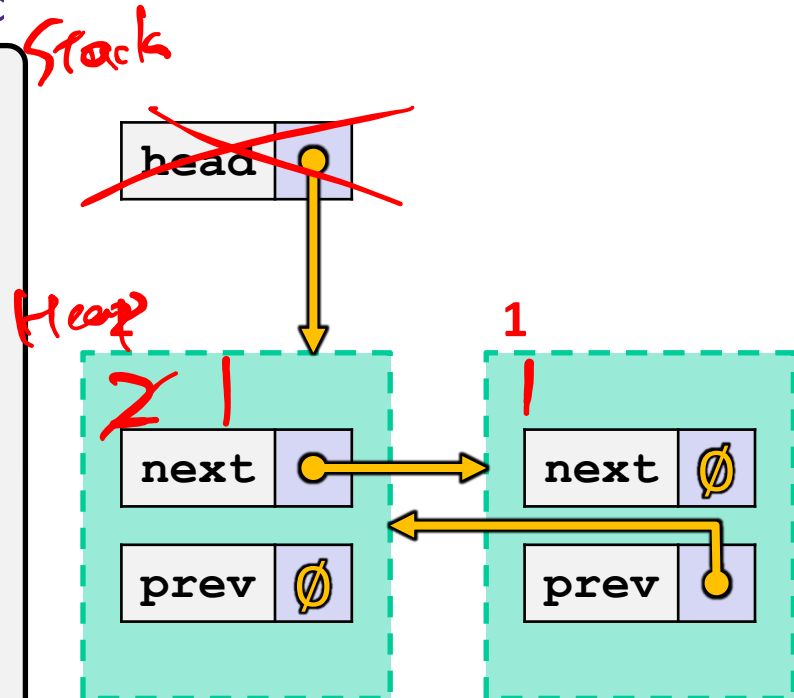
```
#include <cstdlib>
#include <memory>

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 `main` returns?

`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

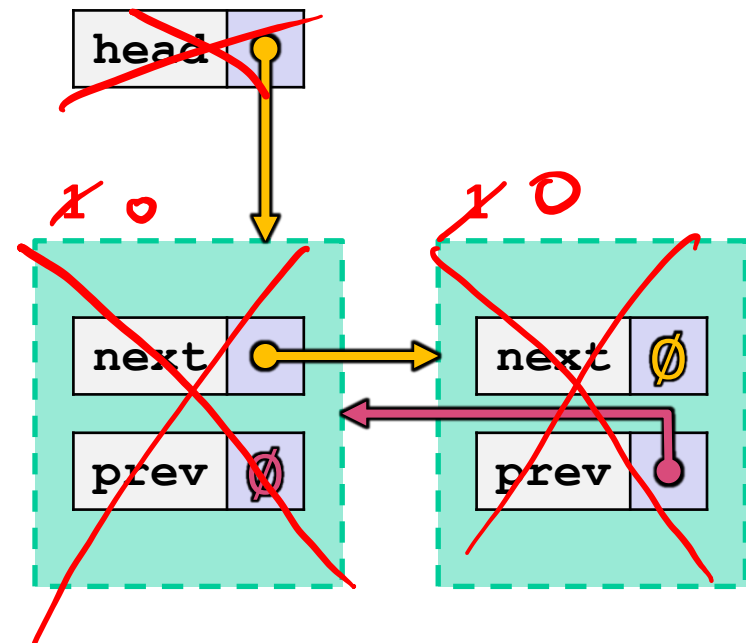
```
#include <cstdlib>
#include <memory>

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 `main` returns?

Be careful: Dangling 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)); y
            w = y;
            x = w.lock(); // returns "promoted" shared_ptr
            std::cout << *x << std::endl; 10
        }
        std::cout << *x << std::endl;
    }
    std::shared_ptr<int> a = w.lock();
    std::cout << a << std::endl;
    return EXIT_SUCCESS;
}

```

The diagram illustrates the state of memory during the execution of the code. It shows three variables: `w`, `x`, and `a`. `w` is a `weak_ptr`, `x` is a `shared_ptr`, and `a` is a `shared_ptr`. A `shared_ptr` `y` points to an integer `10`. `w` is assigned `y`. `x` is assigned `w.lock()`, which returns a promoted `shared_ptr` to `10`. `a` is assigned `w.lock()` after the inner scopes have exited. Red annotations show that `w`, `x`, and `y` are 'Expired!' (dashed boxes with red slashes), while `a` still points to the integer `10`. Handwritten red marks '2', '1', and '0' are next to the integer `10`, and '*a;' is written next to the `a` variable.

Who really owns the managed pointer?

- ❖ Using raw pointers:
 - Recall in HW1 & HW2, we specifically documented who takes ownership of a resource
 - The owner is responsible for calling `free/delete` when it's time to delete the resource
- ❖ Using `shared_ptr`:
 - Never calls `delete` manually, resources are automatically deleted when reference count is 0
 - But when will that happen?
- ❖ It's hard to reason about ownerships of resources when using `shared_ptr`!

Introducing: `unique_ptr`

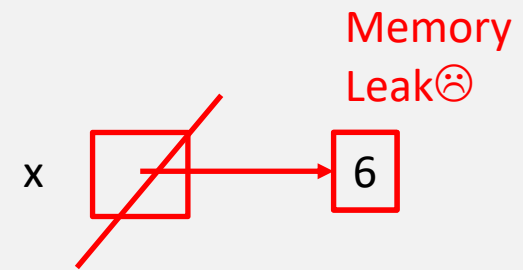
- ❖ A `unique_ptr` is the *sole owner* of its pointee
 - No reference count needed
 - It will call `delete` on the managed pointer when it falls out of scope
- ❖ Enforces uniqueness by disabling copy and assignment

Using `unique_ptr`

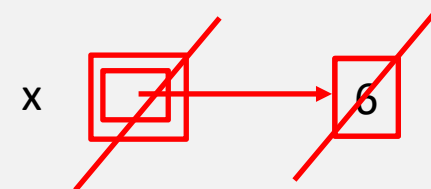
unique1.cc

```
#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;
}
```

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)); // 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;
}
```

Transferring Ownership

unique3.cc

```

int main(int argc, char** argv) {
    unique_ptr<int> x(new int(5));
    // get() returns a pointer to pointed-to object
    cout << "x: " << x.get() << endl;

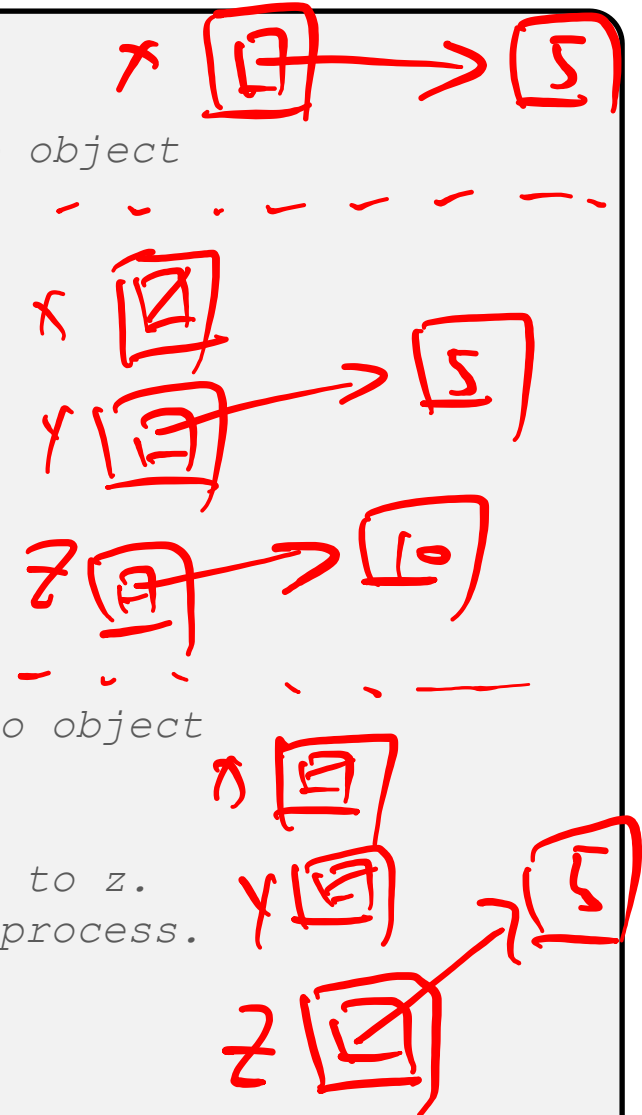
    // Release responsibility for
    // freeing and returns the pointer
    unique_ptr<int> y(x.release());
    // x abdicates ownership to y
    cout << "x: " << x.get() << endl;
    cout << "y: " << y.get() << endl;

    unique_ptr<int> z(new int(10));

    // reset() deallocate current pointed-to object
    // and store new pointer
    // Combined effect:
    // 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;
}

```

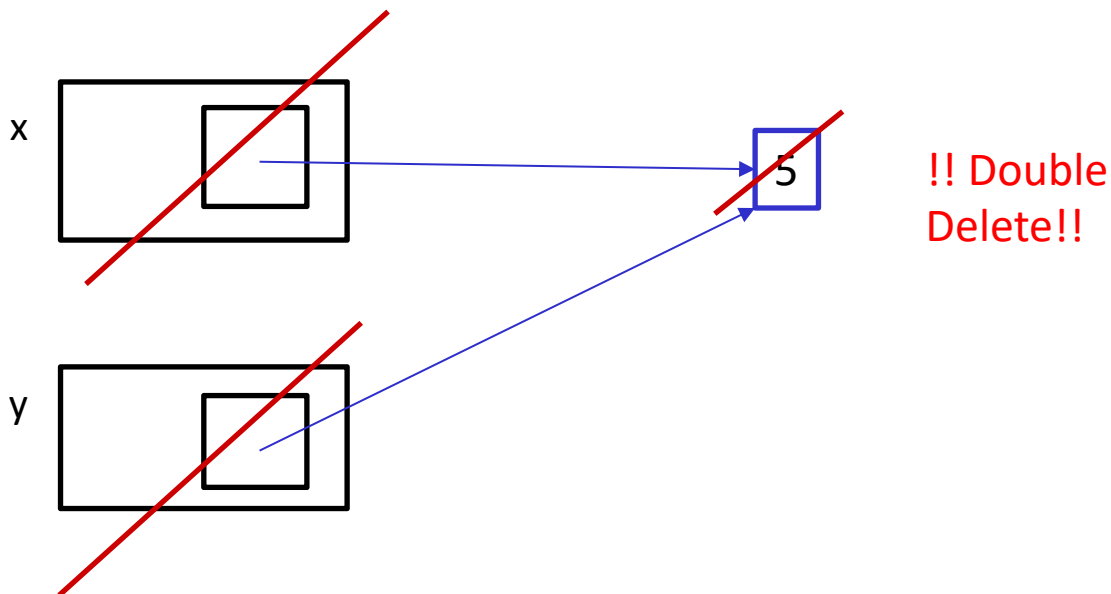


Caution with get() !!

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;
}
```



unique_ptr and STL

- ❖ `unique_ptr`s *can* also 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

```
std::string ReturnString(void) {  
    std::string x("Hello");  
    return x; // this return might copy  
}  
  
int main(int argc, char** argv) {  
    std::string a("World");  
    std::string b(a); // copy a into b  
  
    b = ReturnString(); // copy return value into b  
  
    return EXIT_SUCCESS;  
}
```

copysemantics.cc

Aside: Move Semantics (C++11)

- ❖ “Move semantics”
move values from one object to another without copying (“stealing”)
 - Useful for optimizing away temporary copies
 - A complex topic that uses things called “*rvalue references*”
 - Mostly beyond the scope of 333 this quarter

movesemantics.cc

```
std::string ReturnString(void) {  
    std::string x("Hello");  
    // this return might copy  
    return x;  
}  
  
int main(int argc, char **argv) {  
    std::string a("World");  
  
    // 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(ReturnString());  
    std::cout << "b: " << b << std::endl;  
    return EXIT_SUCCESS;  
}
```

unique_ptr and STL Example

uniquevec.cc

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

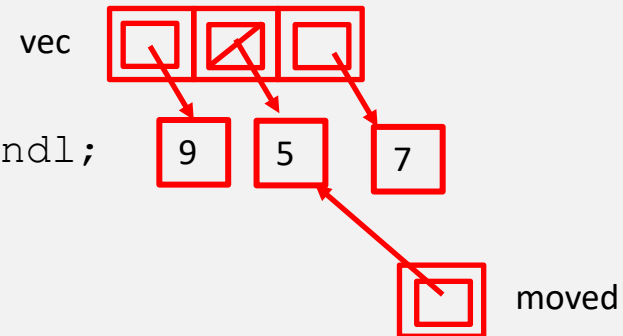
    vec.push_back(std::unique_ptr<int>(new int(9)));
    vec.push_back(std::unique_ptr<int>(new int(5)));
    vec.push_back(std::unique_ptr<int>(new int(7)));

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

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

    // moved points to 5, vec[1] is nullptr
    std::unique_ptr<int> moved = std::move(vec[1]);
    std::cout << "*moved: " << *moved << std::endl;
    std::cout << "vec[1].get(): " << vec[1].get() << std::endl;

    return EXIT_SUCCESS;
}
```



“Smart” Pointers

- ❖ Smart pointers still don't know everything, you have to be careful with what pointers you give it to manage

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;

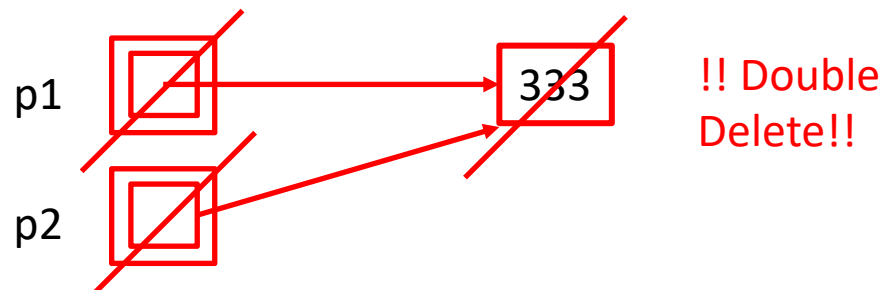
int main(int argc, char** argv) {
    int* x = new int(333);

    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>

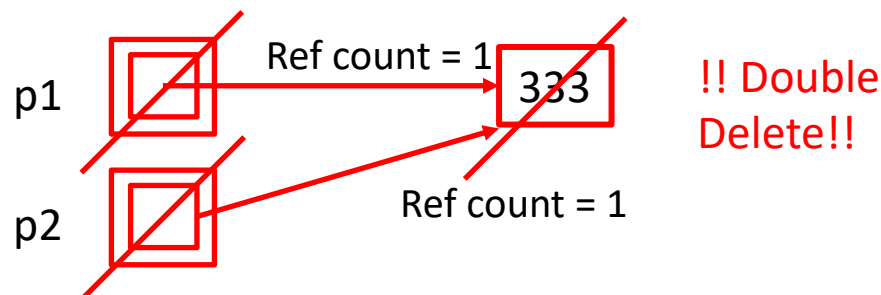
using std::shared_ptr;

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

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

    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;
}
```

- ❖ 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!

Smart Pointers and Arrays

- ❖ `unique_ptr` and `shared_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]); // same for shared_ptr
    x[0] = 1;
    x[2] = 2;

    return EXIT_SUCCESS;
}
```

Summary

- ❖ 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, 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
 - `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

Some Smart Pointer Methods

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

- ❖ `std::unique_ptr 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 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 W;`
 - `W.lock()` Constructs a shared pointer based off of W and returns it
 - `W.use_count()` Returns the reference count
 - `W.expired()` Returns true iff W is expired (`W.use_count() == 0`)