

Smart Pointers

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat

Jialin Cai

Mati Davis

Donglun He

Chandravarman Kunjeti

Heyi Liu

Shufan Liu

Eddy Yang

Logistics

- ❖ Mid Semester Survey **Due TONIGHT 3/20 @ 11:59 pm**
 - Graded, but all submissions get 100% credit

- ❖ Check-in 06 Due Wednesday 3/22 @ 10 am

- ❖ Project details & HW3 to be released soon-ish
 - Project to be done in pairs

- ❖ Grades
 - Midterm grading is being worked on
 - HW2 grades are in progress
still missing submissions from about 1/4th of the class

Lecture Outline

❖ Smart Pointers

- Intro and `toy_ptr`
- `unique_ptr`
- Reference Counting and `shared_ptr` vs `weak_ptr`

In Previous Lectures...

- ❖ Objects
 - ❖ Templates
 - ❖ Destructors
 - ❖ Memory management
-
- ❖ All of these relate to “Smart” Pointers

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 that is 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 frees the pointer
 - Overloaded * and -> operators that access the pointer

*A smart pointer is just a
Template object.*

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
    Takes advantage of implicit calling of destructor to clean up for us
    T &operator*() { return *ptr_; }       // * operator
    T *operator->() { return ptr_; }       // -> operator

private:
    T *ptr_;                               // the pointer itself
};

#endif // _TOYPTR_H_
```

ToyPtr Example

usetoy.cc

```

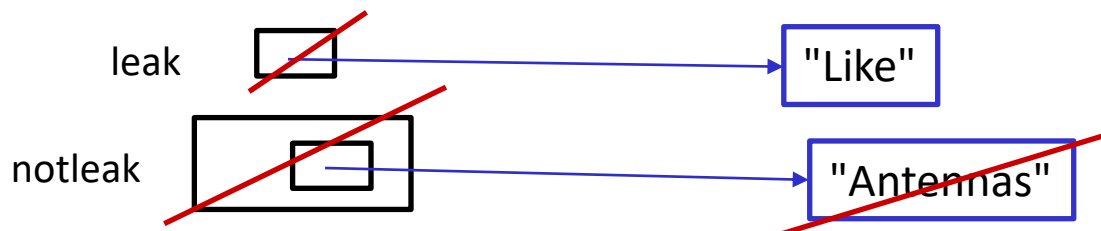
#include <iostream>
#include <cstdlib>
#include "ToyPtr.h"

int main(int argc, char **argv) {
    // Create a dumb pointer
    std::string *leak = new std::string("Like");

    // Create a "smart" pointer (OK, it's still pretty dumb)
    ToyPtr<std::string> notleak(new std::string("Antennas"));

    std::cout << "    *leak: " << *leak << std::endl;
    std::cout << "*notleak: " << *notleak << std::endl;

    return EXIT_SUCCESS;
}
    
```



Notleak cleans up for us,
leak has a memory leak

What Makes This a Toy?

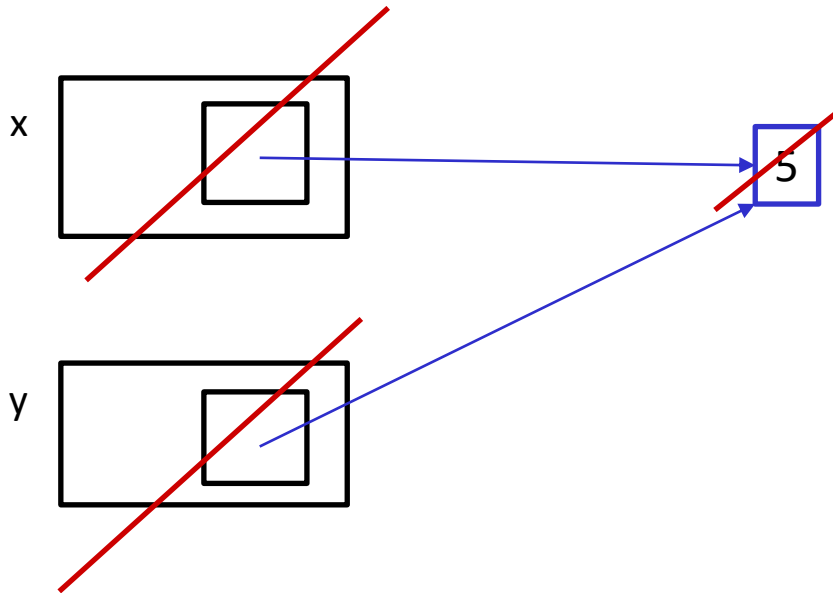
- ❖ Can't handle:
 - Arrays *// needs to use delete []*
 - Copying
 - Reassignment
 - Comparison
 - ... plus many other subtleties...
- ❖ Luckily, others have built non-toy smart pointers for us!

ToyPtr Class Template Issues

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;
}
```



!! Double Delete!!

Lecture Outline

❖ **Smart Pointers**

- Intro and `toy_ptr`
- **`unique_ptr`**
- Reference Counting and `shared_ptr` vs `weak_ptr`

Introducing: `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
Via the `unique_ptr` destructor
- ❖ Guarantees uniqueness by disabling copy and assignment

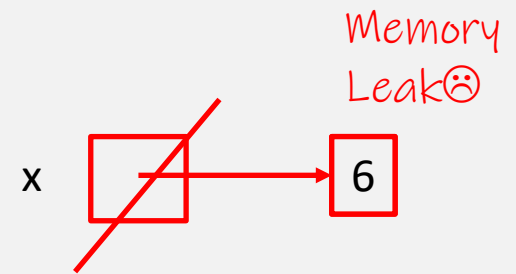
Using `unique_ptr`

unique1.cc

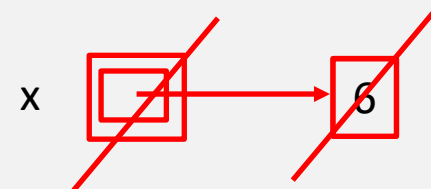
Must include `<memory>`

```
#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_ptr 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;
}
    
```



yes

Compiles



no

Doesn't compile

unique_ptr Operations

unique2.cc

```
#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;
```

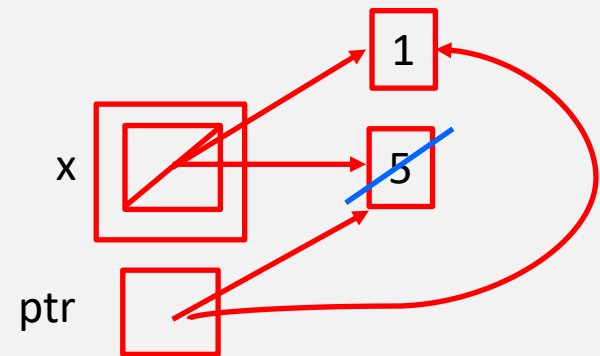
```
    // Deallocate current pointed-to object and store new pointer
    x.reset(new int(1));
```

```
    ptr = x.release(); // Release responsibility for freeing
```

```
    delete ptr;
```

```
    return EXIT_SUCCESS;
```

```
}
```



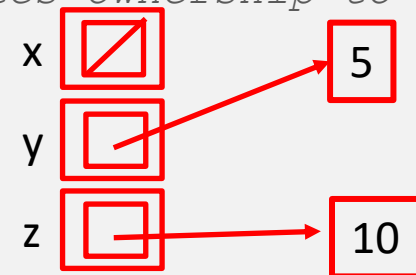
Transferring Ownership

- ❖ Use **reset()** and **release()** to transfer ownership
 - **release** returns the pointer, sets wrapped pointer to `nullptr`
 - **reset** **delete**'s the current pointer and stores a new one

```
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;
cout << "y: " << y.get() << endl;
```



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

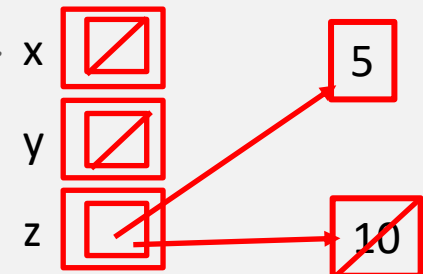
```
// y transfers ownership of its pointer to z.
```

```
// z's old pointer was delete'd in the process.
```

```
z.reset(y.release());
```

```
return EXIT_SUCCESS;
```

```
}
```

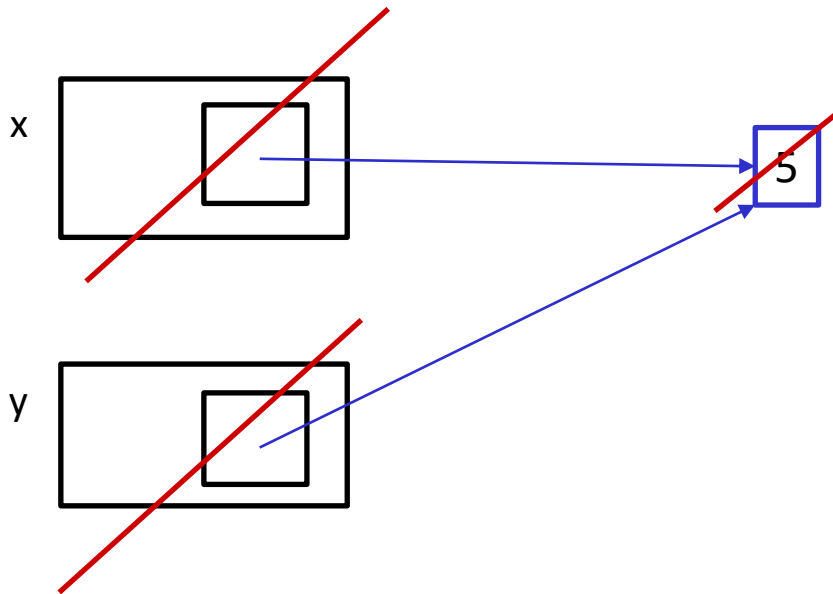


Caution with get() !!

UniqueGetFail.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;
}
```



!! Double Delete!!



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

We will discuss move semantics briefly, not enough time to talk about deeply

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("Quack");
    return x; // this return might copy
}

int main(int argc, char **argv) {
    std::string a("bleg");
    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)

movesemantics.cc

- ❖ “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 this class

```

std::string ReturnString(void) {
    std::string x("Quack");
    // this return might copy
    return x;
}

int main(int argc, char **argv) {
    std::string a("bleg");

    // 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;
}
    
```

a: ""
b: "bleg"

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)));

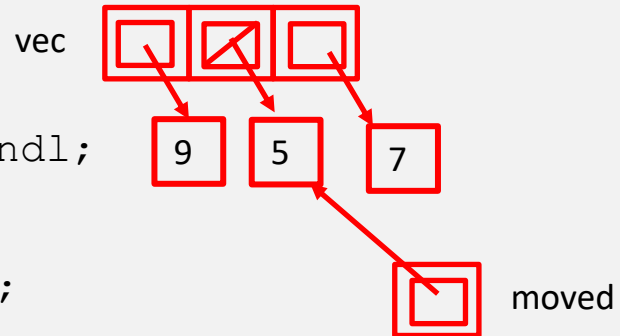
    // z holds 5
    int z = *vec[1];
    std::cout << "z is: " << z << std::endl;

    // 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;
}

```



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

❖ **Smart Pointers**

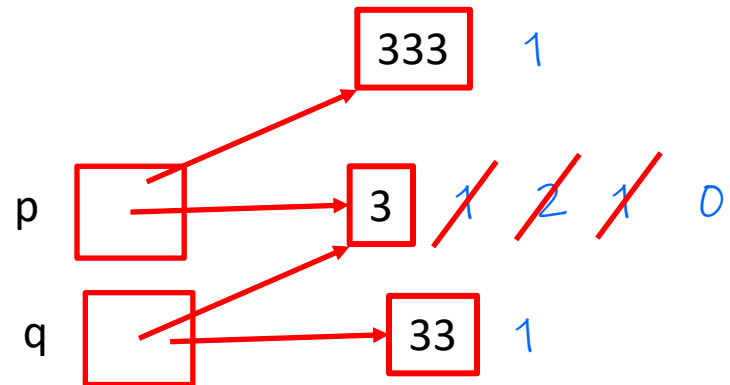
- Intro and `toy_ptr`
- `unique_ptr`
- **Reference Counting and `shared_ptr` vs `weak_ptr`**

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

```

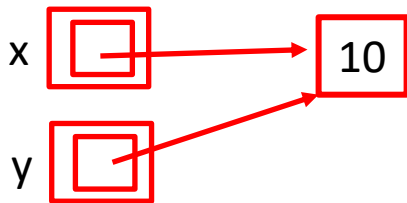
#include <cstdlib> // for EXIT_SUCCESS
#include <iostream> // for std::cout, std::endl
#include <memory> // for std::shared_ptr

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_ptr and STL Containers

- ❖ Even simpler than `unique_ptr`
 - Safe to store `shared_ptr` 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

```

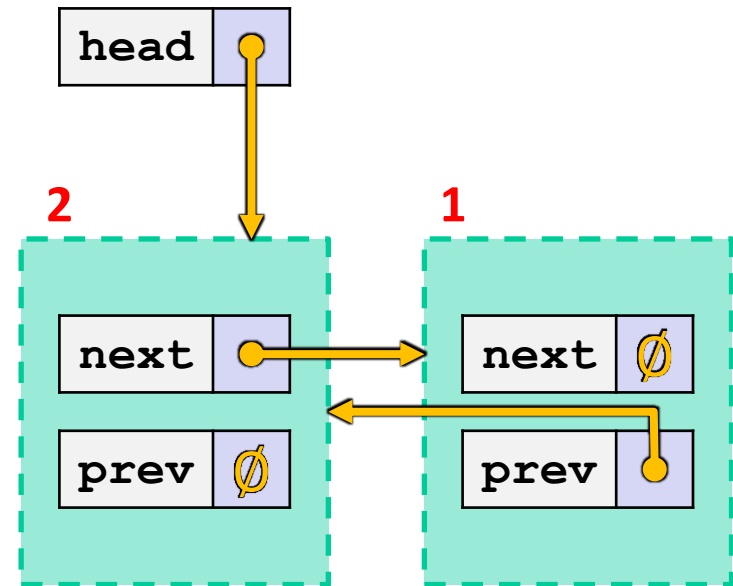
#include <cstdlib>
#include <memory>

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

```

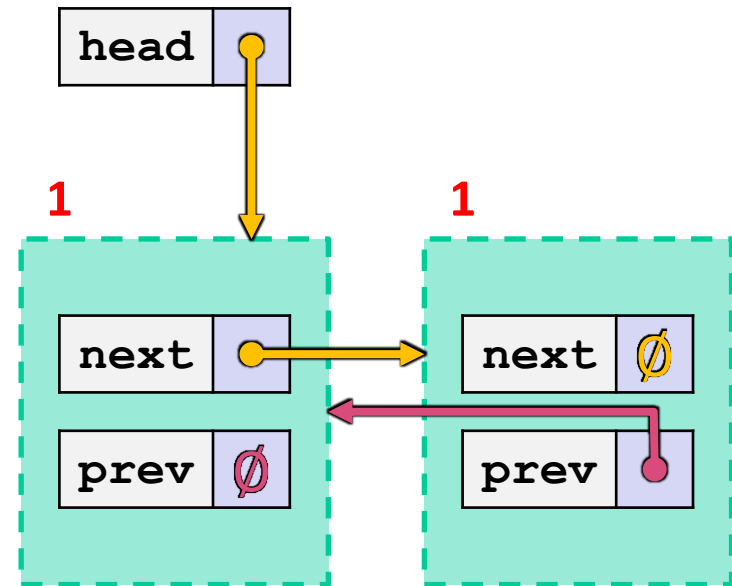
#include <cstdlib>
#include <memory>

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)); y
            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;
}

```

The diagram illustrates the state of the program at the end of the execution. It shows three variables: `w`, `x`, and `p2`. `w` is a `weak_ptr` (represented by a dashed red box), `x` is a `shared_ptr` (represented by a solid red box), and `p2` is a `shared_ptr` (represented by a solid red box). A memory box containing the value 10 is shown. A solid red arrow points from `x` to the memory box, and another solid red arrow points from `p2` to the memory box. A dashed red arrow points from `w` to the memory box, labeled "Expired!". All three variables (`w`, `x`, and `p2`) are shown with a blue diagonal line through their boxes, indicating they are no longer active or have been destroyed.

“Smart” Pointers

- ❖ Smart pointers still don't know everything, you must be careful with what pointers you give it to manage.
 - Smart pointers can't tell if a pointer is on the heap or not.
 - Still uses delete on default.
 - Smart pointers can't tell if you are re-using a raw pointer.

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;

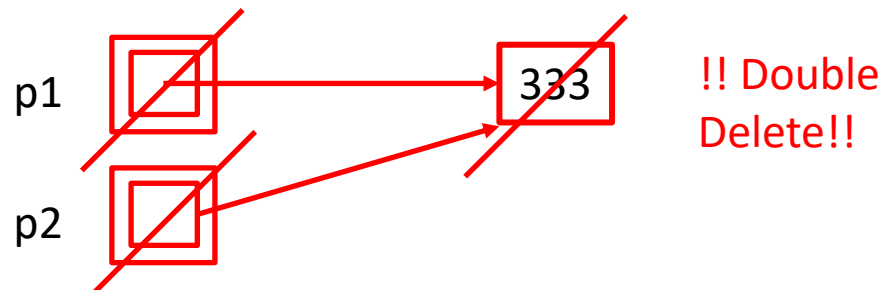
int main(int argc, char **argv) {
    int *x = new int(333);

    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>

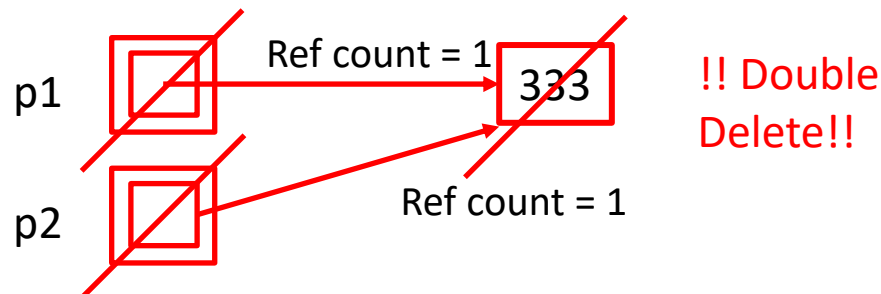
using std::shared_ptr;

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 ctor
 - To be extra safe, don't have a raw pointer variable!

Lecture 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()` `deletes` old pointer value and stores a new one
- ❖ A `shared_ptr` allows shared objects to have multiple owners by doing *reference counting*
 - `deletes` 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

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`)