

# CSE 333

## Lecture 14 -- smart pointers

**Hal Perkins**

Paul G. Allen School of Computer Science & Engineering

University of Washington

# Administrivia

New exercise out today, due Wednesday morning

Exam Friday in class

- Closed book, no notes — exam questions can be more straightforward that way; reference info on test as needed
- Topics: everything from lectures, exercises, project, etc. up to HW2 & basics of C++ (including references, const, classes, constructors, destructors, new/delete, templates/STL, but not smart pointers)
- Old exams and topic list on the web now
- Review in sections this week

# Administrivia (Wednesday)

## Exam Friday, in class

- Closed book, no notes — exam questions can be more straightforward that way; reference info on test as needed
- Topics: everything from lectures, exercises, project, etc. up to HW2 & basics of C++ (including references, const, classes, constructors, destructors, new/delete, templates/STL, but not smart pointers)
- Old exams and topic list on the web now
- Review in sections tomorrow

Unique pointer exercise out Friday, due Monday morning

HW3 out now; brief demo in class today; due in two weeks

Do we need office hours this Friday?

# Administrivia

## Upcoming topics

- finishing up C++ (smart pointers then subclasses)
- rest of quarter: networking, tools, more systems topics, other good stuff

# Last time

## We learned about STL

- noticed that STL was doing an enormous amount of copying
- we were tempted to use pointers instead of objects
  - ▶ but tricky to know who is responsible for delete'ing and when

# 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
  - ▶ how? by overloading `*` , `->` , `[]` , etc.
- a smart pointer 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
  - ▶ so, if you use a smart pointer correctly, 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

see `toyptr/`

# What makes it a toy?

Can't handle:

- arrays
- copying
- reassignment
- comparison
- ...plus many other subtleties...

Luckily, others have built non-toy smart pointers for us!



# C++11's `std::unique_ptr`

The `unique_ptr` template is part of C++'s standard library

- available starting with the C++11 standard

A `unique_ptr` **takes ownership** of a pointer

- when the `unique_ptr` object is *delete*'d or falls out of scope, its destructor is invoked, just like any C++ object
- this destructor invokes `delete` on the owned pointer

# Using a unique\_ptr

```
#include <iostream> // for std::cout, std::endl
#include <memory>    // for std::unique_ptr
#include <stdlib.h>  // 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;
}
```

unique1.cc

# 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**

```
int NotLeaky() {
    std::unique_ptr<int> x(new int(5));

    lots of code, including several returns
    lots of code, including a potential exception throw
    lots of code

    return 1;
}
```

# unique\_ptr operations

```
#include <memory>    // for std::unique_ptr
#include <stdlib.h>   // 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));

    // Return a pointer to the pointed-to object
    int *ptr = x.get();

    // Return a reference to the pointed-to object
    int val = *x;

    // Access a field or function of a pointed-to object
    unique_ptr<IntPair> ip(new IntPair);
    ip->a = 100;

    // Deallocate the pointed-to object and reset the unique_ptr with
    // a new heap-allocated object.
    x.reset(new int(1));

    // Release responsibility for freeing the pointed-to object.
    ptr = x.release();
    delete ptr;
    return EXIT_SUCCESS;
}
```

# unique\_ptrs cannot be copied

std::unique\_ptr  
disallows the use of its  
copy constructor and  
assignment operator

- therefore, you cannot copy a unique\_ptr
- this is what it means for it to be “unique”

```
#include <memory>
#include <stdlib.h>

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

    // fail, no copy constructor
    std::unique_ptr<int> y(x);

    // succeed, z starts with NULL pointer
    std::unique_ptr<int> z;

    // fail, no assignment operator
    z = x;

    return EXIT_SUCCESS;
}
uniquefail.cc
```

# Transferring ownership

You can use `reset()` and `release()`

- `release()` returns the pointer, sets wrapper's pointer to NULL
- `reset()` delete's the current pointer, acquires 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()); // y takes ownership, x abdicates it
    cout << "x: " << x.get() << endl;
    cout << "y: " << y.get() << endl;

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

    // z delete's its old pointer and takes ownership of y's pointer.
    // y abdicates its ownership.
    z.reset(y.release());

    return EXIT_SUCCESS;
}
```

# Copy semantics

Assigning values typically means making a copy

- sometimes this is what you want
  - ▶ assigning the value of one string to another makes a copy
- sometimes this is wasteful
  - ▶ returning a string and assigning it makes a copy, even though the returned string is ephemeral

```
#include <iostream>
#include <string>

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

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

    // copy return value into b.
    b = ReturnFoo();

    return EXIT_SUCCESS;
}
```

copysemantics.cc

# Move semantics

C++11 introduces “move semantics”

- moves values from one object to another without copying (“steal”)
  - ▶ Source object likely modified to “remove” old data
- useful for optimizing away temporary copies
- complex topic
  - ▶ “rvalue references”
  - ▶ mostly beyond scope of 333 (this qtr anyway)

```
#include <iostream>
#include <string>

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

movesemantics.cc



# Move semantics and `unique_ptr`

## `unique_ptr` supports move semantics

- can “move” ownership from one `unique_ptr` to another
- old owner:
  - ▶ post-move, its wrapped pointer is set to `NULL`
- new owner:
  - ▶ pre-move, its wrapped pointer is delete'd
  - ▶ post-move, its wrapped pointer is the moved pointer

# Transferring ownership

## Using move semantics

```
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); // y takes ownership, x abdicates it
    cout << "x: " << x.get() << endl;
    cout << "y: " << y.get() << endl;

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

    // z delete's its old pointer and takes ownership of y's pointer.
    // y abdicates its ownership.
    z = std::move(y);

    return EXIT_SUCCESS;
}
```

unique4.cc

# unique\_ptr and STL

unique\_ptrs can be stored in STL containers!!

- but, remember that STL containers like to make lots copies of stored objects
  - and, remember that unique\_ptrs cannot be copied
  - how can this work??

Move semantics to the rescue

- when supported, STL containers will move rather than copy
  - luckily, unique\_ptrs support move semantics

# unique\_ptr and STL

see [uniquevec.cc](http://uniquevec.cc)

# unique\_ptr and “<”

a unique\_ptr implements some comparison operators

- e.g., a unique\_ptr implements the “<” operator
  - ▶ but, it doesn't invoke “<” 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 sort with a comparison function

# unique\_ptr and sorting with STL

see [uniquevecsort.cc](http://uniquevecsort.cc)

# unique\_ptr, “<” and maps

Similarly, you can use unique\_ptrs as keys in a map

- good news: a map internally stores keys in sorted order
  - ▶ so iterating through the map iterates through the keys in order
  - ▶ under the covers, by default, “<” is used to enforce ordering
- bad news: as before you can't count on any meaningful sorted order using “<” of unique\_ptrs
  - ▶ instead, you specify a comparator when constructing the map

unique\_ptr, “<” and maps

see [uniquemap.cc](#)



# unique\_ptr and arrays

unique\_ptr can store arrays as well

- will call delete[] on destruction

```
#include <memory>      // for std::unique_ptr
#include <stdlib.h>    // for EXIT_SUCCESS

using namespace std;

int main(int argc, char **argv) {
    // x is a unique_ptr storing an array of 5 ints
    unique_ptr<int[]> x(new int[5]);

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

    return EXIT_SUCCESS;
}
```

unique5.cc

# C++11 has more smart ptrs

## **shared\_ptr**

- copyable, reference counted ownership of objects / arrays
- multiple owners have pointers to a shared object

## **weak\_ptr**

- similar to shared\_ptr, but doesn't count towards refcount

# shared\_ptr

A `std::shared_ptr` is similar to a `std::unique_ptr`

- but, the copy / assign operators increment a reference count rather than transferring ownership
  - ▶ after 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 zero, the pointed-to object is deleted

# shared\_ptr example

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

int main(int argc, char **argv) {
    // x contains a pointer to an int and has reference count 1.
    std::shared_ptr<int> x(new int(10));

    {
        // x and y now share the same pointer to an int, and they
        // share the reference count; the count is 2.
        std::shared_ptr<int> y = x;
        std::cout << *y << std::endl;
    }
    // y fell out of scope and was destroyed. Therefore, the
    // reference count, which was previously seen by both x and y,
    // but now is seen only by x, is decremented to 1.
    std::cout << *x << std::endl;

    return EXIT_SUCCESS;
}
```

sharedexample.cc

# shared\_ptrs and STL containers

Even simpler than `unique_ptr`

- safe to store `shared_ptr` in containers, since copy/assign maintain a shared reference count and pointer

see `sharedvec.cc`

# weak\_ptr

If you used `shared_ptr` and have a cycle in the sharing graph, the reference count will never hit zero

- a `weak_ptr` is something like a `shared_ptr`, but it doesn't count towards the reference count and doesn't support `*`, `->`
- a `weak_ptr` breaks the cycle
  - ▶ but, a `weak_ptr` can become dangling
    - but, can check `weak_ptr` to see if dangling
    - and, if `weak_ptr`'s object still exists (not dangling), can construct a regular `shared_ptr` from it to access the object

# cycle of shared\_ptr's

```
#include <memory>

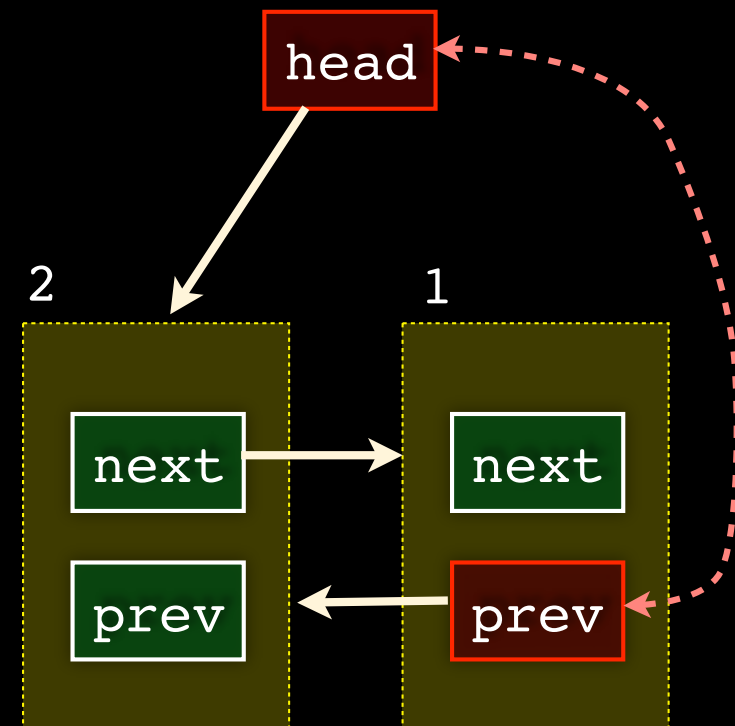
using std::shared_ptr;

class A {
public:
    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 0;
}
```

strongcycle.cc



# breaking the cycle with weak\_ptr

```
#include <memory>

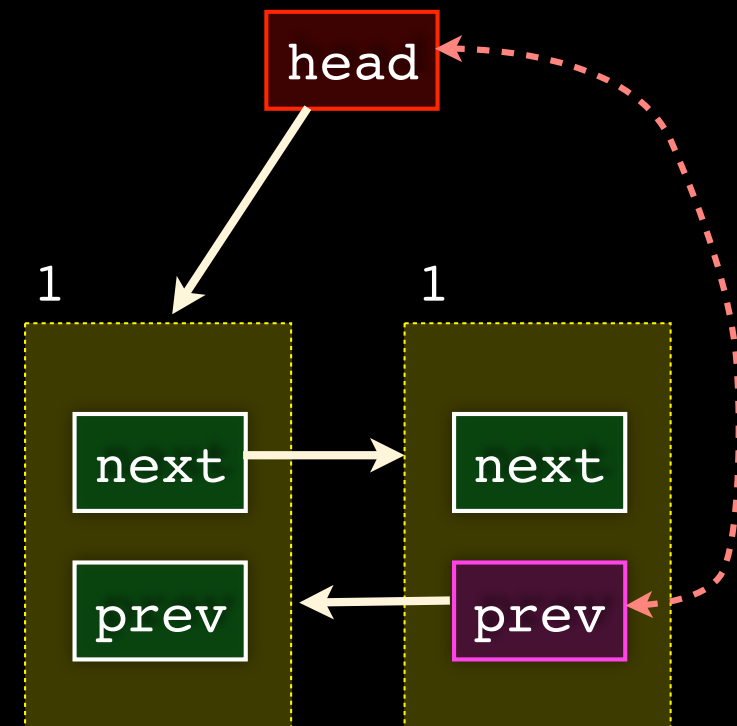
using std::shared_ptr;
using std::weak_ptr;

class A {
public:
    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 0;
}
```

weakcycle.cc





# using a weak\_ptr

```
#include <iostream>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

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

    {
        shared_ptr<int> x;
        {
            shared_ptr<int> y(new int(10));
            w = y;
            x = w.lock();
            std::cout << *x << std::endl;
        }
        std::cout << *x << std::endl;
    }
    shared_ptr<int> a = w.lock();
    std::cout << a << std::endl;
    return 0;
}
```

usingweak.cc

# Exercise 1

Write a C++ program that:

- has a Base class called “Query” that contains a list of strings
  - ▶ (Feel free to wait until after we’ve talked about C++ subclasses)
- has a Derived class called “PhrasedQuery” that adds a list of phrases (a phrase is a set of strings within quotation marks)
- uses a `shared_ptr` to create a list of Queries
- populates the list with a mixture of Query and PhrasedQuery objects
- prints all of the queries in the list

# Exercise 2

Implement Triple, a templated class that contains three “things.” In other words, it should behave like `std::pair`, but it should hold three objects instead of two.

- instantiate several Triple that contains `shared_ptr<int>`'s
- insert the Triples into a vector
- reverse the vector

See you on Friday!

But wait!! There's more!!!