CSE 333
Lecture 13 -- smart pointers

Hal Perkins
Department of Computer Science & Engineering
University of Washington
Administrivia

HW2 is due Tomorrow!

Midterm Monday

- Topic list posted later today
  ‣ Everything in lectures (including basic C++), exercises, hw project
  ‣ Study hint: do lecture exercises, make up sample problems — don’t just “go over” the slides

- Review in sections tomorrow

- Additional review Q&A Sunday, 2 pm, CSE 403

Back to regular exercises and HW3 right after midterm
Last time

We learned about STL

- noticed that STL was doing an enormous amount of copying
- 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
    - 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
C++’s auto_ptr

The auto_ptr class is part of C++’s standard library

- it’s useful, simple, but limited

  • gradually being replaced by alternatives in C++11 - but still used

- an auto_ptr object takes ownership of a pointer

  ‣ when the auto_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 an `auto_ptr`

```cpp
#include <iostream>  // for std::cout, std::endl
#include <memory>    // for std::auto_ptr
#include <stdlib.h>  // for EXIT_SUCCESS

void Leaky() {
    int *x = new int(5); // heap allocated
    (*x)++;
    std::cout << *x << std::endl;
}

void NotLeaky() {
    std::auto_ptr<int> x(new int(5)); // wrapped, heap-allocated
    (*x)++;
    std::cout << *x << std::endl;
}

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

If you have many potential exits out of a function, it’s easy to forget to call `delete` on all of them

- `auto_ptr` will delete its pointer when it falls out of scope
- thus, an auto_ptr also helps with exception safety

```cpp
int NotLeaky() {
    std::auto_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;
}
```
auto_ptr operations

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

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

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

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

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

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

    // Reset the auto_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;
}
```
Transferring ownership

The copy and assignment operators transfer ownership
- the RHS auto_ptr’s pointer is set to NULL
- the LHS auto_ptr’s pointer now owns the pointer

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

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

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

    // z deletes its old pointer and takes ownership of y's pointer.
    // y abdicates its ownership.
    z = y;

    return EXIT_SUCCESS;
}
```
auto_ptr and STL

auto_ptrs cannot be used with STL containers :(  
- a container may make copies of contained objects  
  ‣ e.g., when you sort a vector, the quicksort pivot is a copy  
- accessors will unwittingly NULL-ify the contained auto_ptr

```cpp
void foo() {
    vector<auto_ptr<int>> ivec;
    ivec.push_back(auto_ptr<int>(new int(5)));
    ivec.push_back(auto_ptr<int>(new int(6)));
    // might make copies

    // Accessing a vector element makes a copy of it; therefore, this
    // transfers ownership out of the vector
    auto_ptr<int> z = ivec[0];  // ivec[0] now contains a NULL auto_ptr
}
```
auto_ptr and arrays

STL has no auto_ptr for arrays

- an auto_ptr always calls delete on its pointer, never delete[ ]
Community supported, peer-reviewed, portable C++ libraries

- more containers, asynchronous I/O support, statistics, math, graph algorithms, image processing, regular expressions, serialization/marshalling, threading, and more

Already installed on attu, ugrad workstations, CSE VMs

- or, you can download and install from:
  ‣ http://www.boost.org/
Boost smart pointers

The Boost library contains six variations of smart pointers

- `scoped_ptr` : non-transferrable ownership of a single object
- `scoped_array` : non-transferrable ownership of an array
- `shared_ptr` : shared, reference-counted ownership
- `shared_array` : same as `shared_ptr`, but for an array
- `weak_ptr` : similar to `shared_ptr`, but doesn’t count towards the reference count
- `intrusive_ptr` : we won’t discuss in 333
Sidebar: C++11

In 2011, an updated standard for C++ was approved

- includes a number of enhancements to the language and to the standard library
- one of them is to incorporate into the standard libraries some of the smart pointers that Boost pioneered
  - another is to deprecate auto_ptr!

We’ll stick to Boost for now

- the compilers are racing to adopt the new C++11 features
- should be in place any month year now?
**boost::scoped_ptr**

`scoped_ptr` is similar to `auto_ptr`
- but a `scoped_ptr` doesn’t support copy or assignment
  ‣ therefore, you cannot transfer ownership of a `scoped_ptr`
  ‣ and therefore, you cannot use one with STL containers

Intended to be used to manage memory within a scope
- connotes that the managed resource is limited to some context

*C++2011 has std::unique_ptr, which is very similar*
#include <boost/scoped_ptr.hpp>
#include <stdlib.h>

class MyClass {
public:
    MyClass(int *p) { sptr_ = p; }
private:
    // A MyClass object’s sptr_ resource is freed when the object’s
    // destructor fires.
    boost::scoped_ptr<int> sptr_;}

int main(int argc, char **argv) {
    // x’s resource is freed when main() exits.
    boost::scoped_ptr<int> x(new int(10));

    int *sevenptr = new int(7);
    MyClass mc(sevenptr);

    return EXIT_SUCCESS;}
scoped_array

Identical to scoped_ptr, but owns an array, not a pointer

```cpp
#include <boost/scoped_array.hpp>
#include <stdlib.h>

int main(int argc, char **argv) {
    boost::scoped_array<int> x(new int[10]);
    x[0] = 1;
    x[1] = 2;
    return EXIT_SUCCESS;
}
```
shared_ptr

A shared_ptr is similar to an auto_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

```cpp
#include <iostream>
#include <boost/shared_ptr.hpp>
#include <stdlib.h>

int main(int argc, char **argv) {
    // x contains a pointer to an int and has reference count 1.
    boost::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.
        boost::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;
}
```
shared_ptrs and STL containers

Finally, something that works!

- it is safe to store shared_ptrs in containers, since copy/assign maintain a shared reference count and pointer

but, what about ordering?

- a map is implemented as a binary tree
  - therefore, it needs to order elements
  - therefore, it needs elements to support the “<“ operator

- similarly, what about sorting a vector of shared_ptr<int>’s?
shared_ptr and “<“

A shared_ptr implements some comparison operators:

- e.g., a shared_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)
- this means you can use shared_ptrs as keys in maps
  - but you can’t count on any meaningful sorted order, unless you provide a comparison function in the constructor
- similarly, to use sort on vectors, you have to provide sort with a comparison function
Example

```cpp
bool sortfunction(shared_ptr<int> x, shared_ptr<int> y) {
    return *x < *y;
}

bool printfunction(shared_ptr<int> x) {
    std::cout << *x << std::endl;
}

int main(int argc, char **argv) {
    vector<shared_ptr<int> > vec;

    vec.push_back(shared_ptr<int>(new int(9)));
    vec.push_back(shared_ptr<int>(new int(5)));
    vec.push_back(shared_ptr<int>(new int(7)));

    std::sort(vec.begin(), vec.end(), &sortfunction);
    std::for_each(vec.begin(), vec.end(), &printfunction);
    return EXIT_SUCCESS;
}
```

sharedsort.cc
Putting it all together

see alltogether/
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 just like a shared_ptr, but it doesn’t count towards the reference count
- a weak_ptr breaks the cycle
  ‣ but, a weak_ptr can become dangling
#include <boost/shared_ptr.hpp>

using boost::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;
    }
}
breaking the cycle with weak_ptr

```cpp
#include <boost/shared_ptr.hpp>
#include <boost/weak_ptr.hpp>
using boost::shared_ptr;
using boost::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

```cpp
#include <boost/shared_ptr.hpp>
#include <boost/weak_ptr.hpp>
#include <iostream>

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

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

Exercise 1

Write a C++ program that:

- has a Base class called “Query” that contains a list of strings
- has a Derived class called “PhrasedQuery” that adds a list of phrases (a phrase is a set of strings within quotation marks)
- uses a Boost 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 Monday!