Object-Oriented Programming

CSCI-UA 0470-001
Class 22
Instructor: Randy Shepherd

Smart Pointers

Smart Pointers

- A smart pointer is a C++ class that mimics a regular pointer in syntax and some semantics, but it does more.

- With smart pointers we can some of what the Java garbage collector does. (i.e. deallocate memory)

- Our smart pointer will do *reference counting*, as discussed last class.

Ptr

- The Ptr wrapper class which we used as an example for operator overloading will be enhanced to provided reference counting.
- An implementation of this is in the java-lang-5 repository.
- One core question, though,if we want to track the reference count of every object, where in memory do we store that information?

Reference Count

- Do we store the count as an instance variable *inside the data layout* of our Ptr? *(i.e. internal containment)*
- That has problems…
    - When two instances of Ptr point to the same underlying object and one of them goes out of the scope, both counters in two Ptrs need to be decremented.
    - Clunky and difficult to access internal state of separate instance from a destructor.
    - Clunky and difficult to avoid double deletes.

Reference Count

- How about on the heap with a reference to it as an instance variable? (*i.e. external containment*)
  - With this approach count can be shared across all Ptr’s that wrap some object instance.
  - All we have to do is do a shallow copy of Ptr there is an assignment or a copy.
  - Moreover, the count lives on the heap and is shared by all copies of a particular instance of a Ptr.

Smart Pointer Constructor

- When first initializing a Ptr, construct it with a pointer to a counter on the heap. (line 3)

- On each copy we initialize the copy with a pointer to the same counter on the heap. (line 10)

```
1  /* size_t is a type used for storing unsigned integer
2     independentof architectures.  */
3  Ptr(T* addr = 0) : addr(addr), counter(new size_t(1)) {
4    TRACE(addr);
5  }
6
7  /* copy constructor for Ptr<U> */
8  template<typename U>
9  Ptr(const Ptr<U>& other)
10   : addr((T*)other.addr), counter(other.counter) {
11   TRACE(addr);
12   ++(*counter);
13 }
```

Smart Pointer Destructor

- On destruction we check to see if the reference count it at 0, if so, deallocate. (lines 3-6)
- Depending on the type of the *pointee,* we have to do a different type of delete.
- Therefore ‘delete policies’ are introduced. (lines 9-28)
- (The delete policy is a type parameter given when initializing a ptr.)

```
1  ~Ptr() {
2    TRACE(addr);
3    if (0 == --(*counter)) {
4      delete_policy::destroy(addr);;
5      delete counter;
6    }
7  }
8
9  template<typename T>
10 struct object_policy {
11     static void destroy(T* addr) {
12       delete addr;
13     }
14 };
15
16 template<typename T>
17 struct array_policy {
18     static void destroy(T* addr) {
19       delete[] addr;
20     }
21 };
22
23 template<typename T>
24 struct java_policy {
25     static void destroy(T* addr) {
26       if (0 != addr) addr->__vptr->__delete(addr);
27     }
28 };
```

Smart Pointer Destructor

- For normal C++ objects, simply call delete. This is our 'object_policy' (line 9)

- For arrays, we need to make sure that all memory is freed at each array offset using the delete[] command. This is our 'array_policy' (line 16)

```
1  ~Ptr() {
2    TRACE(addr);
3    if (0 == --(*counter)) {
4      delete_policy::destroy(addr);;
5      delete counter;
6    }
7  }
8
9  template<typename T>
10 struct object_policy {
11     static void destroy(T* addr) {
12       delete addr;
13     }
14 };
15
16 template<typename T>
17 struct array_policy {
18     static void destroy(T* addr) {
19       delete[] addr;
20     }
21 };
22
23 template<typename T>
24 struct java_policy {
25     static void destroy(T* addr) {
26       if (0 != addr) addr->__vptr->__delete(addr);
27     }
28 };
```

Smart Pointer Destructor

- The classes in java-lang may be arrays or objects, therefore we have to add a delete method into our vtable.
- As a result, each java::lang type knows how to clean up after itself.
- This is our java_policy. (line 23)

```
1  ~Ptr() {
2    TRACE(addr);
3    if (0 == --(*counter)) {
4      delete_policy::destroy(addr);;
5      delete counter;
6    }
7  }
8
9  template<typename T>
10 struct object_policy {
11     static void destroy(T* addr) {
12       delete addr;
13     }
14 };
15
16 template<typename T>
17 struct array_policy {
18     static void destroy(T* addr) {
19       delete[] addr;
20     }
21 };
22
23 template<typename T>
24 struct java_policy {
25     static void destroy(T* addr) {
26       if (0 != addr) addr->__vptr->__delete(addr);
27     }
28 };
```

Smart Pointer Assignment

- Whenever we do an assignment we do our normal self-assignment checks, but also increment or decrement the counter.
- Again, since we are passing a pointer to a counter around, we can share the count across instances.
- Note that we use the correct delete policy here as well.

```
Ptr& operator=(const Ptr& right) {
  TRACE(addr);
  if (addr != right.addr) {
    if (0 == --(*counter)) {
      TRACE("cleanup");
      delete_policy::destroy(addr);
      delete counter;
    }
    addr = right.addr;
    counter = right.counter;
    ++(*counter);
  }
  return *this;
}
```

Team Exercise

- The best way to understand it is to start using it.
- Get into your teams. You know the drill.
- The repo https://github.com/nyu-oop-fall16/smart-pointer-in-class