C++: Smart Pointers

Apology

- Office Hours yesterday

Announcement

- RIT Career Fair
  - Thursday, Oct 3rd 1pm – 7pm
  - Friday, Oct 4th 9am – 5pm (interviews)
- Clark Gym
- www.rit.edu/co-op/careers

Announcement

- Date for Exam 1 has been changed!
  - New date: Monday Oct 7th
  - Topics:
    - UML
    - C++ Basics: classes, variables, datatypes
    - C++ Adv: const, static, constructors, operators
    - Memory Management

Project

- Questions?
- Everyone have a partner?
- Please e-mail me with the name of your partner and I will assign you a group account.
- **Design diagrams due tonight !!!!**

Plan for today

- Case Study: Smart Pointers

Memory Leak

- A bug in a program that prevents it from freeing up memory that it no longer needs.
- As a result, the program grabs more and more memory until it finally crashes because there is no more memory left.
- In short:
  - Allocating without cleaning up.

Pointer Ownership

- Everything that is a pointer should be owned
  - Responsible for cleanup when finished
  - Should be known to programmer
  - Should be by design during implementation.
  - Owner and only owner should perform a delete.

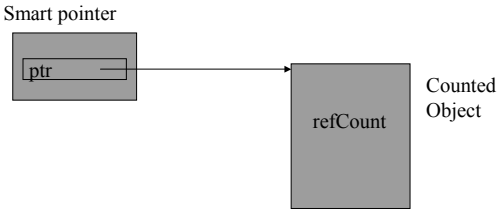
Dangling Pointers

- Pointer is pointing to something that it shouldn't be.
- Can happen if:
  - If the scope of a pointer extends beyond that of the object being pointed to
    - i.e Returning a pointer to a local variable.
  - If a dynamically allocated memory cell is freed explicitly and then the pointer pointing to such a space is used in subsequent code.

Getting around these problems

- The smart pointer
  - Prevents memory leaks and dangling pointers
  - Wrapper class that owns a pointer to an object
    - Object keeps a reference count of variables accessing it
    - When the reference count reaches 0, the object is deleted.
    - After deleting object, pointer value set to 0.

The Smart Pointer



The Smart Pointer

```
class SmartPointer
{
private:
    CountedObject *myPtr;

public:
    SmartPointer (CountedObject *p);
}


SmartPointer P(new CountedObject());
```

The Smart Pointer

- The smart pointer should look and act like a regular pointer:

```
SmartPointer P(new CountedObject());

P->foo(); // should do the same as
// cptr->foo();
```

The Smart Pointer

- This can be achieved by overloading the -> operator

```
class SmartPointer
{
private:
    CountedObject *myPtr;

public:
    SmartPointer (CountedObject *p);
    CountedObject * operator->();
}
```

The Smart Pointer

- This can be achieved by overloading the -> operator

```
CountedObject * SmartPointer::operator->()
{
   return myPtr;
}

SmartPointer P(new CountedObject());

 P->foo();  // P-> returns pointer to a Counted
            // Object
```

The Smart Pointer

- Maintaining a reference count
  - The reference count indicates how many smart pointers are assigned to a pointer variable.
    - `SmartPointer P(new CountedObject());  // ref = 1`
    - `SmartPointer Q = P;  // copy constructor ref = 2`
    - `SmartPointer R;`
    - `R = Q                    // operator=  ref = 3`
    - `P = SmartPointer(new CountedObject());// ref count of what P was`
    - `                         // point to = 2`

The Smart Pointer

- Maintaining the Reference Count
  - The reference count will change when:
    - Smart pointer is constructed
    - Smart pointer is copy constructed
    - Smart pointer is assigned to a new SmartPointer.
    - SmartPointer's destructor is called..

The Smart Pointer

- Maintaining the Reference Count
  - Meaning the smart pointer must define
    - Constructor
    - Copy constructor
    - operator=
    - Destructor
  - Note: the smart pointer is a `friend` to the class of pointers that it is managing.

The Smart Pointer

- Constructors

```
SmartPointer::SmartPointer( CountedObject *ptr ):
   myPtr(ptr)
{
   myPtr->refCount++;
}

SmartPointer::SmartPointer( SmartPointer &c ):
   myPtr (c.myPtr)
{
   if ( myPtr != 0 ) myPtr->refCount++;
}
```

The Smart Pointer

- Assignment

```
const SmartPointer &SmartPointer::operator=
    (const SmartPointer &c )
{
    if ( myPtr != c.myPtr ) {
        if ( myPtr != 0 ) {
            // we're no longer referencing what we were
            myPtr->refCount--;
            if (myPtr->refCount == 0) {delete myPtr; myPtr=0;}
        }
        if ( c.myPtr != 0 ) {
            // it now has one more reference
            c.myPtr->refCount++;
        }
        myPtr = c.myPtr;
     }
 return *this;
```

The Smart Pointer

- Destructor

```
SmartPointer::~SmartPointer()
{
    if ( myPtr != 0 ) {
       // we now have one less reference
      myPtr->refCount --;

       // If we now have no refs, delete it.
       if (myPtr->refCOunt == 0) {
          delete myPtr;
          myPtr = 0;
    }
}
```

The Smart Pointer

- Questions?

Smart Pointers and You

- In your project:
  - You might decide to create an abstract Configuration class.
  - You'll need to maintain a queue and possibly a Map of Configurations.
    - Using STL Maps and Queues of course.

Smart Pointers and You

- Funny thing about C++ Inheritance
  - You can only gain polymorphic behavior on pointers (or references) to objects an not on objects themselves.

```
Vehicle V = Car ();   // not allowed
Vehicle *V = new Car (); // okay
```

Smart Pointers and You

- Back to the project
  - In order to take advantage of polymorphism in C++
    - You'll need to maintain a queue and possibly a Map of Pointers to Configurations.
    - Who is going to be the owner of these pointers?
    - One solution, maintain queues and maps of SmartPointers to configurations.
    - SmartPointers are the keeper of the Configuration Pointers.

Smart Pointers and You

- Smart Pointer code for your project:
  - http://www.cs.rit.edu/~cs4/pub/SmartPointer
  - CountedObject is Configuration
  - SmartPointer is ConfigurationPointer
  - Look at code…
    - good use of assertions to test pre and post conditions

Smart Pointers and You

- Another problem with maintaining maps of pointers to configurations.
  - These STL classes will compare the items placed in them
    - Meaning that actual pointer values (memory addresses) will be compared
    - What we would like instead is to compare the objects being pointed to.
    - Smart pointers can be extended to provide this.

Smart Pointers and You

- In fact…
  - The smart pointers provided to you do!

```
bool ConfigurationPointer::operator<
      ( const ConfigurationPointer &c ) const
{ return *_config < *c._config; }
```

Smart Pointers and You

- Questions?

Generic Smart Pointers

- Our Smart Pointers
  - Have been "hard coded" for the project.
  - Generic Smart Pointers use Templates to indicate what the smart pointer is pointing to.

```
template T
class SmartPointer {
...
}

SmartPointer<Foo> f;   // smart pointer to Foo objects
```

Summary

- Smart Pointers
  - Wrapper class on a pointer
  - Takes ownership of pointer
    - Reduce memory leaks
    - Reduce dangling pointer
  - Comparison operator for use in STL classes.