"Smart" Pointers and Standard C++ vector Class

Topics Covered

- What are "smart" pointers
- Overloading the indirection operator
- Overloading the pointer-to-member operator
- Smart pointers and the copy constructor
- The C++ auto_ptr class
- Introduction to the Standard Template Library
- The vector template class

What are "Smart" Pointers I

- Why do pointers create problems in programs?
  - Because they have two "meanings" [roles]
  - 1. Pointers often act as "iterators" for objects that already exist, such as this string copy
     ```cpp
     while (*dest++ = *src++) ;
     ```
  - 2. Pointers also act as "monikers" [nicknames] for independent objects created with new
     ```cpp
     Cat * p = new Cat;
     ```
What are "Smart" Pointers II

- Pointers have "value semantics" as iterators
  - What is meant by "value semantics"?
  - Means objects you can copy and assign to [like int]
    ```
    int x = 3, y = 7, temp;
    temp = x; x = y; y = temp;
    ```
  - Raw pointers [like p below] have value semantics
    ```
    int a[10], *p = a + 10;
    while ( p-- != a ) cout << *p << endl;
    ```

What are "Smart" Pointers III

- Pointers used as monikers act different
  ```
  Cat *p = new Cat;
  ```
  - Not only points to a Cat, it owns the Cat
  - You can easily lose that ownership [resource leak]
    ```
    p = 0;
    ```
  - You also cannot freely assign without catastrophe
    ```
    Cat * q = p; delete p; delete q;
    ```

What are "Smart" Pointers IV

- Smart pointers are C++ objects that:
  - "Mimic" regular pointers by overloading:
    » The indirection [dereferencing] operator
    » The pointer-to-member operator
    » Other operators depending upon role
      » Comparison and increment often added
  - Provide value semantics [copy constructor]
    - Provide some kind of "ownership management"
      » Usually a built-in destructor
Smart pointers are usually template classes
- The class T is the type of value you'll point to
- Usually have a T* as a data member
- Destructor automatically cleans up memory

```cpp
template <class T> class ptr
{
    T* p;
public:
    explicit ptr(T* pv= 0) : p(pv) { }
    ~ptr() { delete p; }
}
```

Things to notice about the ptr class
- Default constructor creates a NULL pointer
  » Eliminates problem of uninitialized pointers
- Only good for creating objects with new
  » Destructor automatically deletes object
  » Don't point to stack-based or static variable
- Cannot use as a pointer to dynamic array
  » Uses delete, not delete[]

Raw pointers are prone to resource leaks
for (int i = 0; i < 3; i++)
    int * p = new int(rand())); // Leaks here

Smart pointers automatically delete heap memory
for (int i = 0; i < 3; i++)
    ptr<int> p(new int(rand())); //OK

Must explicitly call the constructor
The Indirection Operator

- Used to dereference the smart pointer [SmartPointer02]
  - Returns a reference to object pointed to
  - Dereference “internal pointer” and return that
    - T& operator*() { return *p; }
  - Use to access “pointee” just like regular pointer
    - // Read value from memory
    - cout << “p = “ << *p << endl;
    - // And write values to memory
    - "p = rand();"

The Member Selection Operator

- Calling member functions [SmartPointer03]
  - Override operator->(), return the raw pointer value
    - T* operator->() { return p; }
  - Acts differently than other overloaded operators
    - C++ recursively applies operator->()
      - ptr<Fish> p(new Fish);
      - p->swim();
      - // Translated as : p.operator->()->swim();

Copy and Assignment

- What should happen when you write this?
  - ptr<string> p1(new string("First");
  - ptr<string> p2(p1);

- Who knows? With smart pointers its up to you
  - There are three basic possibilities
  - Each is preferable under certain circumstances
    - Deep copy [cloning]
    - Reference count [tracking]
    - Destructive copy [transfer]
Simple Deep Copy

- Clones the pointed-to object

```cpp
explicit ptr(const ptr & rhs)
: p(new T(*rhs.p)) {};
```

- Some disadvantages
  - Slower than reference counting
  - Doesn’t work in presence of polymorphism

Polymorphic Deep Copy

- Only way to handle copy of polymorphic pointers
  - Can’t do this with raw pointers

```cpp
Shape * a[2] = { new Circle(...), new Square(...) };
ar[0] = ar[1]; // How? What “type” is ar[1]?
```

- Create a pure-virtual Clone() function in base
  - Must return a Base *
  - Override in child: return new Circle(*this)
  - Call rhs.p->Clone() to make copy of object

Reference Counting

- Popular, yet most complex method
  - Used for "high-ticket" [expensive to build] items
    - Used extensively in COM programming for instance
  - Used where "pointee" not changed
  - If "pointee" can be changed, then use COW version

- Requires separate counter & object classes
  - Counter incremented in ctor, decremented in dtor
  - Actual object deleted only when count == 0

- See Meyers handout for complete details
Destructive Copy

- Destructive copy "transfers" ownership \[\text{SmartPtr06}\]
  - Old pointer is destroyed when pointer is copied
  - Only one pointer can "own" object at any one time

\[
\text{ptr\& operator=(ptr\& rhs) } \\
\quad \{ \\
\quad \quad \text{if ( \( \text{this} \neq \text{rhs} \) ) } \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad \text{\( p = \text{rhs}.p; \text{rhs}.p = 0; \) \quad // Destroy \text{rhs} \} \\
\quad \quad \} \\
\quad \text{return \text{this};} \\
\quad \}
\]
- Note that rhs is NOT const

The auto_ptr Class

- Std C++ auto_ptr class uses destructive copy
  - See SmartPtr07.cpp example
  - Note that VC++6 uses 1996 [v2] auto_ptr class
  - Uses a "reference" bit, but old values still OK
  - New version uses Std C++ auto_ptr

\[
\text{#include <memory>} \\
\text{auto\_ptr\_int\_new\_int(5), p2;} \\
\text{p2 = p1;} \\
\text{cout << *p1 << endl;} \quad \text{// ILLEGAL} \\
\text{cout << *p2 << endl;} \quad \text{// ILLEGAL}
\]

Introducing the Standard Library

- The C++ Standard Library contains classes and concepts based upon the STL
  - STL : the Standard Template Library
    - Developed by Stepanov, Musser, & Lee
    - http://www.stlport.org/resources/StepanovUSA.html
    - Still under development at SGI?
  - Based upon "generic programming"
    - Develop an algorithm
    - Generalize to work on variety data
The STL is based upon a generic "concept"

- A concept is composed of requirements
  - Requirements not programming language
- An implementation of concept is a "model"
  - Similar to ADT: interface vs implementation

STL has four basic concepts, many models

- Container, iterator, algorithm, functor

What is an STL container?

- An object that stores other objects
  - All objects must be of the same type
  - Data stored in a container is "owned" by the container
  - Expires when container expires
- Objects stored in a container must be
  - Assignable and copy-constructable
- No order assumed by container

STL has several container refinements

- Sequence [linear] containers
  - A first and last element
  - Remaining elements have one next, previous
  - Sequence models: vector, list, deque
  - Adapters: queue, stack, priority_queue
- Associative containers
  - Unordered, with key-value pairs
  - Models: set, multiset, map, multimap
The vector Class

- The vector class is the Std C++ dynamic array
  - Can hold any type of data
  - All items must be the same type
- How to create a vector object [vector01.cpp]

```cpp
#include <vector>

vector<float> vf; // default
vector<int> vi(100); // sized
vector<string> vs(ssize); // variable
```

vector Member Functions I

- Number of elements in a vector
  - `capacity()` is number of elements before reallocation
  - `size()` is number of elements “in use”
    - With vector, both are often the same
  - Can change capacity using `reserve()` method
    - This does not change the `size()` of the array

```cpp
vf.reserve(10);
std::cout << "size = " << vf.size() << "capacity() = " << vf.capacity() << std::endl;
```

- Normally, you’ll use `size()` as array bounds

```cpp
int ilen = vi.size();
for (int i = 0; i < ilen; i++) ...
```

- To access vector elements, you have a choice between safety and speed
  - Use subscript operator if element is in bounds
  - Use `at()` method if you want a range-checked access

```cpp
vi[i] = rand(); // What is i? Possibly unsafe.
vf.at(4) = 3.14; // Range checked
```
Dynamic vectors

- To make a vector "grow", use push_back()
  - Adds an item to end of vector (changes size())
  - If vector is full, then it is automatically resized

Example: [vector2.cpp]
```
vector<float> vf;
vf.reserve(4);
for (int i = 0; i < 10; i++)
  vf.push_back(i * 2.2);
```
- Watch changes in size() and capacity()

Using vector Iterators I

- What is an iterator?
  - A "logical" pointer [pointer concept]
  - Similar to smart pointer without "ownership"
  - Dereference to give you an object of type T
  - Can increment using operator++()

Creating iterator variables
- Use container<type>::iterator as type
```
vector<float>::iterator vi;
```

Traversing a vector

- Every STL container has begin(), end()
  - begin() returns an iterator to first element
  - end() returns an iterator to "past-end" element

```
12  |  3  |  25 |  -6 |  22
   | begin()   |   |   | end() |
```

- Traverse container like this:
```
for ( vi = vf.begin(); vi != vf.end(); vi++)
  cout << *vi << endl;
```
Removing Elements

- Use `erase()` to remove elements in vector
  - Takes a range of two iterators
- Range notation is: `[iter1, iter2)`
  - Not C++ code, just document notation
  - Includes first iterator, up to, not including, second
  - `[iter1, iter1)` is empty range
  - `[begin(), begin() + 2)` is first 2 elements
  - `vf.erase(vf.begin() + 2, vf.end());`

Inserting Elements in a vector

- Use `insert()` to copy partial vectors
  - First argument is iterator object following insertion
    - `vf.begin()` means add in front of first element
    - `vf.end()` means add before "phantom" element
  - Second argument is iterator to first source element
  - Third is iterator to "past-end" in source
  - `vf.insert(vf.end(), // add at the end`  
    `vf2.begin(), // start at 1st element in vf2`  
    `vf2.begin()+3); // copy 3 elements`

Generic Functions I

- Most STL functions are not members
  - Instead, functions can take any container
  - Advantage? function need only be written once
- Example: the `for_each()` generic function
  - Used to replace for loops
  - Supply start and stop iterators, & function
    - `for_each(vf.begin(), vf.end(), print);`
Generic Functions II

- Example: Randomize and sort a vector
  - Use `random_shuffle()` to randomize
  - Only works when container allows random access

```cpp
#include <algorithm>

vector<int> vi;
for (int i = 0; i < 10; i++) vi.push_back(rand());
random_shuffle(vi.begin(), vi.end());
sort(vi.begin(), vi.end());
```

Homework

- Homework 25: C++ strings and vectors
  - Frequency counter: Due Sunday, May 9
    - 1) Create a vector of strings
      - Read each word in file into vector then sort
    - 2) Create a vector of unique words from first
    - 3) Create a vector of counts
      - Process first vector to fill in third
  - Print each unique word and its count