Lecture 22

Miscellaneous Topics 4
+ Memory Allocation

Declarations in Classes: Enumerations
- The constants defined in the enumeration may be used anywhere in the class, and in other classes by the following rules:
  - If the enumerations are declared protected, they may be used only in derived classes.
  - If the enumerations are declared public, they may be accessed from anywhere provided their fully qualified names are used.

```cpp
class Example {
    public:
        void setCounter(int argCounter=kInitialValue);
    private:
        enum CounterValues {
            kInitialValue=0, kLastValue
        }; int counter;
};
```

Declarations in Classes: Static Members
- A static member variable is a variable declared in the class definition but is treated like a global variable:
  - There is only one copy of it no matter how many instances of its class exist
  - If its value is changed by any one instance, the value is changed for all the instances
  - In effect, it is a global variable but only visible outside the class if declared in the public section.

```cpp
int Example::currentTotal = 0;  // Define the static member
```

Static Members
- There’s only one problem...
- Since the static member doesn’t really exist in the class, separate storage must be allocated for it outside the class.
- This is done in the form of a global variable using the fully qualified name of the static member...

```cpp
class Example {
    public:
        Example(); ( currentTotal = 0; )
        void addToTotal(int aValue);
    private:
        static int currentTotal;
};
```

```cpp
int main() {
    Example e1;
    Example::currentTotal = 5;        // There are two ways to cout << e1.currentTotal << endl; // access the same var!
    return 0;
}
```
Static Members
- A static member function is a little different since there is already only one copy of the code used to implement a member function.
- A static member function has no this pointer and cannot access individual non-static member fields.
- However a static member function may be called without creating an instance of the class in which it is defined...

class Example
{
  public:
    static void printBanner() { cout << "Hello World" << endl; }
  
  int main()
  {
    Example::printBanner();
    return 0;
  }
}

Demonstration #1

Declarations in Classes

Nested Classes
- In addition to declaring enumerations and static member variables/functions within classes you can also define other classes within classes.
- These are called "Nested Class Definitions"
- Consider the following:

class X // X is an arbitrary class
{
  public:
    class Y // Y is defined in X's public section
    {
    }
  
  private:
    class Z // Z is defined in X's private section
    {
    }
};

Nested Classes
- The fact that class Y is defined in the public section of class X means the following:
  - Objects of class Y may be created outside the scope of X only if fully qualified name is used (X::Y)
- The fact that class Z is defined in the private section of class X means the following:
  - Objects of class Z may only be created within the scope of X.
  - If Y had been defined within a protected section of class X's definition, objects of type Y could be created anywhere in the scope of X as well as in the scope of any class derived from X.
- Any member variables/functions of Y are accessible according to the normal public, private and protected used in classes.
- Having said all of this, CodeWarrior does not appear to honor the restrictions on Z and will let me declare a class of type X::Z outside of the scope of X.

Nested Classes
- An example of nested class definitions might involve list management.
- Consider the following:

class List
{
  public: // standard constructor
    List():the_head(NULL) {} 
  
  private: // standard destructor
    ~List();
  
  class Node // Nested definition of Node.
  {
    public: // the "value" of an element
      string the_value;
    
    private: // pointer to next element
      Node *next;
      
      Node *the_head; // the first element
  };
Memory Allocation
- No matter what system or platform you are programming for, there is one constant...
- Memory allocation is slow!
- The general problem is that no matter how good the generic memory management algorithm is behind the new operator, it is burdened by the fact that it must be prepared to allocate and manage blocks of memory which are of varying sizes.
- Now, please keep in mind that slow is a relative term. When looked at individually, even the slowest memory allocation is still fairly fast under a modern OS and/or modern hardware.
- Under MacOS, memory management is particularly slow due to there being an extra layer of indirection present (handles).
- So, what a perfect platform to talk about memory management on!

Demonstration #3
Lots of Dynamic Allocation

Memory Allocation
- OK, but that means I actually have to implement some memory management routines!
- WHY would I want to do that?
- Because the generic memory management routines are slow!
- They're slow because they have to be able to deal with any size object.
- If you know you will need to dynamically allocate a lot of the same objects during the course of your program you can define your own memory management routines for allocations of that object type.
- A typical memory management routine to handle repeated allocations of the same type can be implemented in a way that will be more efficient than the generic routine.
- Let's do it!

Memory Management
- The idea is somewhat simple, when you think about it.
- You allocate a block of memory (using the generic routine) to form an "array" of the objects in question.
- When YourObject::operator new is called, you "give out" one of your previously allocated objects.
- That means you need to keep track of what's allocated and what is not!
- When you start out, you'll have something like this:

Memory Management
- But how do you "dole out" pre-allocated objects in an orderly fashion?
- First, you'll need a pointer variable which will be used to point to consecutive items in your array. We'll call it NextFree.
- It will, of course, start at the beginning of the array.
Memory Management
- When one object is allocated via a call to YourObject::operator new, we'll return the object that NextFree is pointing at and then increment NextFree to the next item in the array.
- That would look like this:

```
class Element {
public:
    YourObject theObject;
    Element    *nextElement;
};
```

Memory Management
- But what happens when YourObject::operator delete is called?
- We need to keep track of deleted objects as well.
- We do this in the form of a "pseudo" linked list.
- I say pseudo because in a regular linked list, the implication is that each element is allocated only when necessary.
- What we're going to do is make each object in our pre-allocated array similar to a linked list Node with two fields, the first is an instance of YourObject, the second is a "next" pointer.
- It might look like this:

```
class Element {
public:
    YourObject theObject;
    Element    *nextElement;
};
```

Memory Management
- Now, when we delete an object, we'll put it at the beginning of a linked list of "deleted" objects.
- Whenever YourObject::operator new is called, we'll first check to see if there are any objects on the "deleted objects" list.
- If there are, we'll return the first object from that list instead of the object pointed at by NextFree.
- What happens if we delete that object we just allocated...

```
class Element {
public:
    YourObject theObject;
    Element    *nextElement;
};
```

Memory Management
- OK, now let's say that we allocated three more objects and then delete the second object we allocated in that run.

```
class Element {
public:
    YourObject theObject;
    Element    *nextElement;
};
```

Memory Management
- Now, let's allocate another object.
- We'll use the first one on our list of deleted objects...

```
class Element {
public:
    YourObject theObject;
    Element    *nextElement;
};
```

Memory Management
- But what happens if we allocate all the elements in our array?
- You can always fall back on the default operator new in your YourObject::operator new code.
- As a matter of fact, in YourObject::operator new, you should always check that the size requested matches the size of the object you've defined the memory handler for, and default to ::operator new if it doesn't.
- But what happens in this situation:
Memory Management
- It's actually pretty simple.
- We just allocate another array.
- It doesn’t matter if it’s not contiguous with the previous array, our linked list of deleted objects will mean we’ll never "lose reference" to any object:

<table>
<thead>
<tr>
<th>Deleted</th>
<th>Unallocated</th>
<th>Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NextFree</td>
</tr>
</tbody>
</table>

Memory Management
- Now, let’s allocate two more objects:

<table>
<thead>
<tr>
<th>Deleted</th>
<th>Unallocated</th>
<th>Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NextFree</td>
</tr>
</tbody>
</table>

Memory Management
- And now delete every other object:

<table>
<thead>
<tr>
<th>Deleted</th>
<th>Unallocated</th>
<th>Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NextFree</td>
</tr>
</tbody>
</table>

Memory Management
- Any questions on the memory management algorithm?
- Let’s look at some code:

Demonstration #4
Better Dynamic Allocation?

Lecture 22
Final Thoughts