Managing Dynamic Memory, Reading

Dynamic memory management involves the writing of three functions:

1. a destructor function
2. a copy constructor functions
3. an assignment operator function

Destructor Functions

The destructor function's name is unique like that of the constructor. It's the class' name with a prepended tilde (~) symbol. Now *that's* unique! Like the constructor, there's no return type. Unlike the constructor, there are *no* parameters. Also it's so short that it can be written inline, like this:

```
template <typename V>
class Array
{
  ...

  public:
  Array(int);
  ~Array(){delete [ ] values;}
  ...
};
```

Recall that for every "new [ ]" there needs to be a "delete [ ]" somewhere to balance it and avoid a "memory leak".

Copy Constructors

When we were using static arrays, each copy of a data structure object has its own data array. But now that the array is dynamically-allocated and we store a pointer in the object, we have a big problem with copies -- each copy as the same value for the pointer. That means each copy shares the same array out in the heap! Yes, they're copies, so what difference does it make? Think about it -- if you change the value at an index in one of the copies, the original and all the other copies are affected! Worse, the first object to go out of existence causes its destructor to be called. That deallocates the arrays for all copies and the original. When the second object goes out of existence, it too executes its destructor, but the pointer is no longer valid and the program will crash!

The problem is that copies would share the same pointer values and hence the same arrays. The solution is to find a way for the copy to have its *own* dynamically allocated array, and assign *its* memory location to *its* private data member pointer.

Normally when you do **Array<int> b = a;**, the data members are copied member-by-member. But if you write a "copy constructor function", it intercepts the copy statement and lets the programmer control what gets copied to what. It's prototype is:

```
  Array(const Array<V>&); // prototype
```

When making copies this way, the original constructor *is not called at all.* Only one constructor ever gets called, whether it's the original or the copy. So like the original, the copy's job is to assign the private data members "values" and "cap".

```
template <typename V>
Array<V>::Array(const Array<V>& original)
{
  cap = original.cap; // still just a copy
  values = new V[cap]; // not a copy -- new object gets its own array
  for (int i = 0; i < cap; i++) // copy the contents of the array from the original...
```

```
    values[i] = original.values[i]; // ...to the copy
  dummy = original.dummy; // not really necessary because who cares what dummy contains?
}
```

This function demonstrates an important point about private data members. "original" and the object under construction are different objects. Yes the one under construction appears to have access to the private data members of "original"! How can that be? Well, "private" only applies to code that is *outside* the class. Any code that belongs to the class -- its member functions -- have access to all members of all objects of that class.

Assignment Operators

That's fine for *new* copies, but what about when a copy *replaces* an existing object as in "**Array<int> c; c = a;**". The "equals" operator, also known as the "assignment operator", does a member-by-member copy of each private data member, but it does *not* use the copy constructor because that object "c" was already constructed using the main constructor. Too late -- it's already got memory allocated for its array, and a capacity. So what makes this different from the copy constructor is that the existing array has to be deallocated before the new one can take its place.

Here's its prototype:

```
  Array<V>& operator=(const Array<V>&); // prototype
```

Not as easy as the destructor or copy constructor, because of the return type. At least those other two had no return type, so one less thing to worry about. *But this does*. It returns a *variable* -- a *reference* to an Array object. By convention, the object it should return itself -- it returns a self-reference.

One more thing before we go to code -- it's really subtle. It guards against this possibility: **a = a;**. Of course, we would never write such a statement, but through aliasing and parameters in function calls, it's possible that the "a" and "c" in **c = a;** could be aliases for the same object elsewhere in the code. The reason this is a concern is that the first thing the operator function is supposed to do is deallocate its array. If it then tries to copy the contents of its deallocated array to its newly allocated array, well, that's just not going to work. So we do this:

```
template <typename V>
Array<V>& Array<V>::operator=(const Array<V>& original)
{
  if (this != &original) // of it's not a self copy...
  {
    // do what the destructor does
    delete [ ] values;

    // do what the copy constructor does
    cap = original.cap; // still just a copy
    values = new V[cap]; // not a copy -- new object gets its own array
    for (int i = 0; i < cap; i++) // copy the contents of the array from the original...
      values[i] = original.values[i]; // ...to the copy
    dummy = original.dummy; // not really necessary because who cares what dummy contains?
  }
  return *this; // return a self reference
}
```

The keyword "this" is a pointer to the host object, storing its memory location. **&original** is the memory address of the object being copied over the host. **If those memory addresses are the same**, then it's a *self copy* -- skip it!

***this** is a dereferenced pointer to the host object -- it *is* the host object, which is what this function is supposed to return.