Variable Capacity Data Structures, Reading

In the previous reading we learned how to declare and create a data structure object of a given capacity. That is way better than our previous attempt where the capacity as a number was specified directly in the class' coding. That works fine in many applications, especially since we also learned how to customize the amount of stack memory allocated to our programs.

But it does still have the limitation that we have to *say* that number at the time we write the application and declare and create the object (like **Array<int, 10> a;**). That "10" can be any number or a constant, but it cannot be a variable. For example, our application cannot ask its user how many test scores they wish to enter and create the Array object of just the right capacity.

The STL gets around that problem with the vector (and its close relative, the deque). It does so by putting the capacity in parentheses in the declaration. That number can be a number *or a variable*.

```
vector<int> a(10);

int n = ...
vector<double> b(n);
```

That's a feature we'd like for our Array template to have.

Private Data Members

There are two things we have to do in order to get this to happen. First, we have to use a dynamic array for the private data member instead of a static array, and second, we have to find a way to get that number in parentheses in the declaration statement in main to the class.

First we have to recall how to create a dynamic array. Dynamic arrays use heap memory, which is not from the limited, 1MB stack -- it's from the whole computer system's available memory. With 16GB machines, that's a whole lot more to work with than 1MB! In any case, recall:

```
const int CAP = 10; // capacity as a constant
int a[CAP]; // a static array in stack memory

int cap = ... // capacity as a variable
int* a = new int[cap]; // a dynamic array in heap memory
```

This means that the large array as a private data member gets replaced by a very small, single pointer! All of a sudden, the object size got way smaller. The array is elsewhere in the heap, and it's *associated with* the object, but it's not *inside* the object. Just the pointer is inside, and that's typically 8 bytes. By comparison, our 10-int array was 40 bytes, and that's a really small array.

Since capacity is no longer a constant, we need a private data member to track that, too, so our template changes like this:

```
template <typename V, int CAP>
class Array
{
  V values[CAP];
  ...
};
```

becomes

```
template <typename V>
class Array
{
  V* values;
  int cap;
  ...
};
```

The "Main" Constructor

We still have to find a way to communicate the number in the parentheses in main -- the "n" value in **Array<int> a(n);**) to the class so that it can (1) set

the value of its private data member "cap", and (2) do "new int". That's done through a modification to the constructor, adding a one-parameter parameter list with one int, so the main constructor prototype is now this:

```
Array(int); // prototype -- look for its modification later in this reading!
```

The main constructor's definition now looks like this:

```
template <typename V>
Array<V>::Array(int cap)
{
  ...
}
```

Note that the parameter name matches the private data member name, so we'll need to distinguish them by referring to the data member as **this->cap**.

In the constructor's curly-brace container we need to (1) copy the cap parameter to its matching data member, and (2) allocate the dynamic array of that cap. Like this:

```
this->cap = cap;
values = new V[cap];
```

Note the template code in "new V[ ]" where we would have expected to see "new int[ ]". But remember that the data type for values to be stored in the data structure object is now generalized to V, to be substituted for later on with an actual real data type (possibly int).

The dummy and the data array "values" still need initializing to the **typename V** default value as they did in the previous module.

A Default Constructor

Now that there's a main constructor with an int as its parameter, **Array<int> a;** is no longer possible. There *must be* a set of parentheses with a number inside it in each declaration statement.

One solution to this is to specify a *default* for the parameter in case it's not provided in the declaration statement. It's done like this in the *prototype* (*not* in the function definition):

```
Array(int=2); // 2 is the default capacity
```

With this simple change, it's now possible to write **Array<int> a;,** and if you're main program is not happy with a capacity of 2 (or whatever you choose as the default capacity), then do something like this: **Array<int> a(1000);**.

We've Opened Pandora's Box

It would seem that that's all there is to it -- that we now have something that works like the STL vector and deque. But alas, it's not so simple in C++. In one of those one-thing-leads-to-another cycles, we now have to manage the heap memory allocated to our object. Fortunately, just as there's a function that's called automatically when an object is declared and created, there's another function that's called when an object goes out of existence -- it's the "destructor function". It's the first of a series of functions for managing dynamic memory in objects.