Writing An Array Class, Reading

We've gone about as far as we can with **class Student**. It's not really a data structure, but it demonstrates the coding features that we will be using in our study of data structures. There are a few more programming techniques and more syntax to learn before we're *truly* ready, but they don't make a lot of sense in the context of a Student object. They are a bit more meaningful and useful for data structures, so we'll begin by using what we've learned to create our first data structure, and get into the additional coding features after that's done.

Note that a "data structure" in the context we're studying it, is an *object*. It's an object that contains *like* values -- all ints, all doubles, even other objects like strings and Students. An array already does that, but it's not an object. It has no members that can track its capacity, for instance. In this reading we'll develop a class that **wraps an array in a "class container"** with getters and setters to give it behaviors (like get its capacity) and actions (like set a value at an index).

Specification

A good way to start the development of a data structure, or *any* object for that matter, is to write code samples of how you want it to work. It's a way of writing a "specification". In doing so, many of the names for members, the return types and parameters, get decided. That makes the actual writing of the class easier later.

Here's ours -- it's what we would put in the main program:

```
Array a; // declare and create the object to store 10 int values, all initialized to zero
a.setAtIndex(9, 100); // store the value 100 at index 9, ignore if out of range
cout << a.capacity( ); // get the capacity of the array
cout << a.getAtIndex(8); // get the value stored at index 8, 0 if out of range
```

As you can see from our comments, there will be some checking going on to prevent out-of-range errors. This is called "range-safe" -- our class will be range-safe. This may sound like a no-brainer, but actually there is a price to pay that might not be worth it! Every time we access a value in our data structure, it **has to be checked**. We endure that **overhead** burden even though we are careful programmers and never would try to use an index out of range -- right?

The Private Data Members

Next, we list the private data members for the class. We just need an array of ints of capacity 10. At this point it does not seem that we need any other data members, so we'll stop with this and add more if we need them later.

```
int values[10];
```

The Class Definition

Now we can put the data members into a "class container" along with prototypes for the functions listed in the specification -- note the public interface for the class with its three functions:

```
class Array
{
  int values[10];

  public:
  void setAtIndex(int, int); // index (0-9) and value
  int capacity( ) const {return 10;}
  int getAtIndex(int) const; // index, 0-9
};
```

See how the **coding decisions fall right out of the specification**? The return types, names, and parameters are in the specification. Only the decisions about which are setters and which are getters in not clear from the specification, but they are rather self-explanatory, because two of them (Array::capacity and Array::getAtIndex) have no reason to modify data members of the host object, and are clearly getters. Array::setAtIndex does (attempt to) modify a data member, so it's a setter.

The Function Definitions

One of the functions is so easy, it's written inline (Array::capacity). The other two go below main (or in a separate file):

```
void Array::setAtIndex(int index, int value)
{
  if (index >= 0 && index < 10)
    values[index] = value;
}

int Array::getAtIndex(int index) const
{
  if (index < 0 || index >= 10)
    return 0;
  return values[index]; // no "else" needed
}
```

A "Main Constructor"

There's still the one small matter of initializing all the data values in the array to zero -- zero is the "*default value*" for an int. That can be done in a for-loop, but we need a place to p*ut* that for-loop. We could write an initialization function for the main program to call once the Array object is created, but a better idea is to have such a function called *automatically* every time an Array object is created. For that, C++ classes have what are called "constructor" functions.

The name of the C++ constructor matches the class name. It has ***no return type*** at all, not even void, because it's ***never called directly*** as member functions are called from the main program. It's parameter list is empty -- well, it's empty for now. More on that in future modules.

```
class Array
{
  ...
  public:
  Array( ); // the constructor function's prototype
  ...
};

Array::Array( )
{
  for (int index = 0; index < 10; index++)
    values[index] = 0;
}
```

Because it sets the data member values to their default values, this is called the "*default constructor*" in object-oriented programming.

That's it -- that's the class. But it's *not* how you go about developing the class! Whatever you do, do not just start writing code as we did above. Yes, that's what the final product will look like, but that's *not* how to get it. There's a process for that, and it's part of *program design*. We'll go through that process at the end of this module, but first, there's one more coding feature to introduce in the next reading -- the square bracket operator.