Adjustable Capacity Data Structures, Reading

There's one more thing that the STL has that we don't -- the ability to change the capacity of our array (either or down) after the object has already been created and used. Going back to how we write specifications, this is how we want to use it in main:

```
Array<int> a(100);
a.capacity(200); // change capacity to 200
```

It's obviously a setter. It's got the same name as a getter that *returns* the capacity, but it's parameter distinguishes it from that one.

The function is similar to the assignment operator -- it has to deallocate the array, allocate a new array, and copy the contents. But the order of things has to change a bit because the array to be copied is the array we want to deallocate, so we cannot deallocate until the coping is done.

Here's the algorithm for the process:

1. Allocate a new array of the new capacity, and store its memory location in a *temporary pointer*.
2. Compute a limit for the copying for-loop as the lesser of the old and new capacities.
3. Execute the for-loop to copy the array contents.
4. Execute another for-loop to set any added values to their defaults.
5. Deallocate the original array.
6. Assign the temporary pointer's value to the private data member pointer.
7. Set the private data member capacity to the new capacity.

Here's the prototype:

```
void capacity(int); // setter prototype
```

...and the templated member function:

```
template <typename V>
void Array<V>::capacity(int cap)
{
  // allocate a new array with the new capacity
  V* temp = new V[cap];

  // get the lesser of the new and old capacities
  int limit = min(cap, this->cap); // requires the C++ "algorithm" library

  // copy the contents
  for (int i = 0; i < limit; i++)
    temp[i] = value[i];

  // set added values to their defaults
  for (int i = limit; i < cap; i++)
    temp[i] = V( );

  // deallocate original array
  delete [ ] values;

  // switch newly allocated array into the object
  values = temp;
  // update the capacity
  this->cap = cap;
}
```

The pointer "temp" goes away when the function ends, because it's a local variable. It's the only thing that knows where the newly-allocated array is in heap memory. So before its demise, it "tells" the private data member "values" where that memory is located, so it's not lost after all. But before "values" can change what it points to, it needs to use its original value to deallocate the original array, to avoid a memory leak.