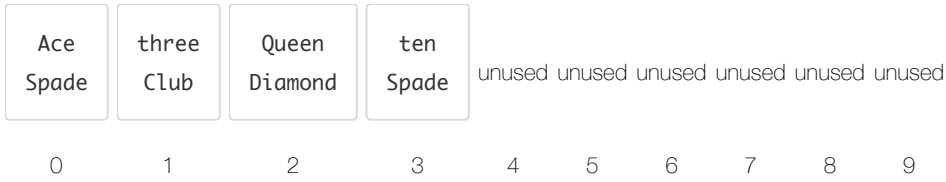
Arrayed Implementation Of A Stack, Reading

In the previous reading we created 4 PlayingCard objects and pushed them into a stack. Using an array to store these PlayingCard objects in a stack object, we'd have something like this (with a capacity of 10):



The *size* of the stack is 4 -- we're going to need a private data member to track that, because it's not the same as the *capacity*. The next unused index in the array is also 4. So in a **push** operation, a new PlayingCard would get copied into the **index that equals the size**, and then the size would be incremented. But before doing so, we'd have to make sure that size and capacity were *not* already equal, which would mean the array was full. In that case, we'd *double the capacity* before proceeding with the push.

The **peek** operation would return a mutable reference to the value at the index equal to **size minus one**. In the example above, size is 4 and the last-added value is at index 3, which is size-1. But if the size is zero, there's nothing to return -- but we still have to return *something* because it's a value-returning function. Since the value at index 0 is not in use, there's no harm in returning that, or we could have a dummy private data member and return that. One or the other...

The **pop** operation, since it returns nothing, simply has to **decrement** the private data member that tracks **size**. There's no need to "delete" the object that's being popped -- we simply ignore it. Remember that this is an array of objects (or other values) and there is no way to have a blank in an array. All those "unused" elements -- they are just uninitialized PlayingCard objects. So as long as size is not already zero, we decrement it, and that's all.

The **clear** operation is an extension of pop, where instead of setting size to itself minus one and ignoring the popped index, we just set size to zero and ignore *everything* in the array. Again, no deleting or zeroing out of objects in the array -- just leave and ignore them.

A Templated Class

We'll do our implementation of a stack as an object, using a template.

```
template <typename V>
class Stack
{
  ...
};
```

Private Data Members

Because of the auto-adjusting capacity, we'll need the **same data members as the DynamicArray**. Plus we'll need an int to track size -- how many indexes are actually in use. But we don't really need the dummy, because we can use index zero for that.

```
  V* values;
  int cap;
  int siz; // track size
```

Notice how the variable names "cap" and "siz" are abbreviated? It's not just to save typing. It's because there are member functions named "capacity" and "size", so the abbreviations avoid name "collisions". (Actually, we don't even *have* **Stack::capacity** in the public interface -- neither as setter or as getter).

Main Constructor

Same as DynamicArray, except that there's one more private data member to **initialize -- set "siz" to zero**.

Same as DynamicArray, except that there's one more private data member to **initialize -- set "siz" to zero**.

```
siz = 0;
```

Copy Constructor And Assignment Operator

Same as DynamicArray, except that these dynamic memory management functions need to **account for the new "siz" data member**, too, setting it to the parameter's value.

```
siz = original.siz;
```

Pop And Clear

These setters are so short they can be written **inline**.

```
void pop( ) {if (siz > 0) --siz;}
void clear( ) {siz = 0;}
```

Push

The push function is something new -- there's no equivalent in DynamicArray. But in Module 3 we had **Array::setAtIndex**, so it will be something like that. The algorithm for push was given earlier in this reading - we end up with this:

```
void push(const V&); // prototype
```

```
template <typename V>
void Stack<V>::push(const V& value)
{
  if (siz == cap) capacity(2 * cap); // double the capacity if full
  values[siz] = value;
  ++siz;
}
```

Peek

The peek function is just like the square bracket operator setter, except that there's *no* parameter. Instead of "index", we use **siz-1**, and instead of **return dummy;** we do **return values[0];**. We could of course continue to use "dummy", if we so chose, but using "dummy" adds just a bit of extra overhead that we don't need.

And like the operator[ ] setter, peek returns a mutable reference so that it's possible for the top value in the stack to be modified from the main program.

```
V& peek( ); // prototype
```

Capacity

Note that the specification does not have a call to the "capacity" functions, either to get the current capacity or to change it. There's no longer a need for the getter, so that should not be included. There is a need for the setter, because the push function uses it. But there's no need for public access to the function.

Here's an opportunity to show how private member functions work. Here's how we keep the "capacity" setter function without letting the main program have access to it:

```
template <typename V>
class Stack
{
  V* values;
  int cap;
  int siz;
  void capacity(int);

  public:
  ...
};
```

That's all there is to it -- you just move its prototype to the private area of the class definition. Another way to code this is to put a **private:** statement after the list of public members, and list the private functions there -- some programmers prefer that way of organizing things in the class definition.