Data Abstraction, Reading

With templated member functions, all of the operations that take place inside the functions *are not specific!* Before we introduced templates, **values[index]=value;** meant to copy an int over an int. It means to copy the bit sequence of the parameter "value" into the memory occupied by "values[index]". Simple.

But in the templated class, it does not mean the same thing. If **typename V** is a string, the bit sequence is *not* simply copied. C++ strings are objects that have dynamic memory. So assignment involves an assignment operator function in the string class.

But *we* know what assignment *means*. How it *does* it depends on what is the thing being copied. Assignment is an *abstraction*. What it means to programming is that the template does not have to know how to do assignment, as long as the data type itself *supports* assignment.

Abstract Data Type, ADT

“V” stands for the data type in a template (or whatever the typename specification calls it). “V” is an abstraction of a data type — an "ADT". Unlike an int or string which are defined by what they *are*, “V” is defined by what it *does*. That is, if there’s an equals-equals in a templated member function that compares two V’s, then “V” is something that can be compared for equality. If there’s a less-than in a templated member function that compares two V’s, then “V” is something that can be compared in terms of order. If there’s an **= 0;** in a templated member function that sets a “V” to zero, then “V” is something that can be set to zero. How it does these things is up to the data type itself. All that’s required of the ADT is that it support the operation.

Why We Do Not Initialize The values Array To Zero

Abstraction is why we don't have a loop in the main constructor that sets the values array elements to zero, like this:

```
~~for (int i = i; i < cap; i++)~~
  ~~values[i] = 0;~~
```

It's because this requires that the **typename V** know how to set itself to zero. It "V" is an int or double, fine -- that works. But if "V" is a string, this will not compile because **string::operator=(int)** is not defined. Strings do not support being set to zero. So we do this instead:

```
for (int i = i; i < cap; i++)
  values[i] = V( );
```

Limitations

We're going to see equals-equals and less-thans in templated member functions in future modules. What that's going to mean is that those functions *will not work* for **typename V**'s that don't define these abstractions.

There must also be a default value defined for **typename V**. If it's a programmer-written struct or class, it better have a default constructor or no constructors at all, or else we can't use it in our templates.