A Templated Array Class, Reading

Let's start with the STL's array, and do what they did in order to make the Array class from our previous module work like that. They applied the "template" keyword, which is best explained by compare and contrast:

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

  public:
  Array( );
  int capacity( ) const {return 10;}
  int operator[ ](int) const;
  int& operator[ ](int);
};
```

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

  public:
  Array( );
  int capacity( ) const {return CAP;}
  V operator[ ](int) const;
  V& operator[ ](int);
};
```

The added line above the "class Array" turns Array into a "templated class". So now we need a name for the class before it became a templated class -- we'll call that a "non-templated class". Non-templated classes are actual code like any other code. The definition and the functions exist and are compiled and are part of the executable program.

But templated classes are *not* actual code. They are *blueprints* for how to create an actual class once the data type name and the int (in the template statement) are specified in a declaration somewhere. When **Array a;** appeared in main in the previous module, the code for Array and all it's functions already existed. But until we the compiler runs into a declaration like **Array<int, 10> a;**, no code exists. It's not until it runs across the declaration that it pulls out the template, substitutes "int" for "V" (which stands for the *values*' data type) and "10" for CAP (short for *capacity*) and automatically generates the code! In fact, until a function of the template is actually called, that function is not generated!

Then if the compiler finds **Array<string, 100> s;** later in the program, it goes back to the template and generates a new version, again including only the functions actually called. If it finds another **Array<int, 10> x;**, no problem -- it's already got that code.

In fact, some compilers don't look at the coding for the functions too carefully, if they are not called in the program being compiled. So it's possible for syntax errors to lurk in templated functions, not to be discovered until later!

Templated Member Functions

Inline functions work as they did before, but functions written outside the class (that is, below main) are affected by this too -- *including the main constructor*. Each function becomes a template, and needs it's own template statement. Again by compare and contrast:

```
int& Array::operator[ ](int index)
{
 if (index < 0) return dummy;
 if (index >= 10) return dummy;
 return values[index];
}
```

```
template <typename V, int CAP>
V& Array<V, CAP>::operator[ ](int index)
{
 if (index < 0) return dummy;
 if (index >= CAP) return dummy;
 return values[index];
}
```

```
int Array::operator[ ](int index) const
{
 if (index < 0) return -999;
```

```
template <typename V, int CAP>
V Array<V, CAP>::operator[ ](int index) const
{
```

```
 if (index >= 10) return -999;
 return values[index];
}
```

```
 if (index < 0) return dummy;
 if (index >= CAP) return dummy;
 return values[index];
}
```

Again, T and CAP get substituted for, based on the Array declaration statement.

Template Syntax

There are some details we glossed over in the above presentation, so let's deal with those now:

1. CAP is a ***constant* int**, so it can be used in declaring static arrays (like "values"). Whatever number that's used in the declaration, that's what's substituted in the class definition and the functions.
2. The **function names** changed. The non-templated member functions were like this: "Array::fun". But the function template names are like "Array<V, CAP>::fun", and the *actual* function names are like "Array<int, 10>::fun" and "Array<string, 100>::fun". But we often just call it "Array::fun" anyway.
3. The "default value" for an int is zero. But that won't work in the general case of **typename V**. It's default value is **V( )**. In fact, **int( )** resolves to zero, the default int value. **int i = int( );** does the same as **int i = 0;**. So your constructor will have something like **dummy = V( );**.

Variations

The template statement has other forms -- some of which we'll see in later modules and readings, and some that you may run across in the work of other programmers. One is the use of that keyword "typename". Some programers prefer its alternate name, "class". Either way, they team exactly the same thing -- only the spelling is different:

```
template <typename V, int CAP>
```

```
template <class T, int N>
```

There's nothing magic about the designation "V". A lot of programmers use "T", which is rather generic. Likewise for CAP -- it's just a name for the const int, and can be any valid C++ identifier.

If there is no need for the second part of the statements -- the int -- then just omit it. And if there's more than one data type involved (and there will be when we get to associative arrays), you can have more than one typename:

```
template <typename T>
```

```
template <typename T, typename U>
```

As long as the data types and/or values in the declaration statements match, that's all that matters.

```
template <typename K, typename V>
class AssociativeArray
{
  ...
};

int main( )
{
  AssociativeArray<string, int> phoneBook;
  ...
}
```

Capacity Limitations

The allowable capacity of the private data member "values" array is limited by the number of bytes of stack memory allocated to your program when it starts. In today's desktop-based systems with modern operating systems like Windows and OS X, 1 megabyte is typically the default. That means that most programs start to top out at about 100,000 capacity for ints and doubles, and much less for objects. That's usually not a problem (although it will be when we get to the module on "big data"), but there's a way to override the **1MB default** in the command line. Here's an example with a **100MB stack** (that's a *w-el*, not a *w-one* by the way):

```
prompt> c++ -std=c++11 -Wl,-stack_size,0x6400000 myProgram.cpp -Wall
```

Note the `0x` -- that means the `6400000` is *base-16*, *hexadecimal*. There's a useful online calculator for getting the base-16 equivalent of any decimal value you may wish to use for stack size -- **www.binaryhexconverter.com/decimal-to-hex-converter (http://www.binaryhexconverter.com/decimal-to-hex-converter)** . Use 1024_{10} for a kilobyte and 1024_{10} x 1024_{10} for a megabyte. Or just ask Siri...

IDEs also have project settings where you can override the default stack size for the program.