Template Header Files, Reading

Before the Array class became templated, and had to be customized (for data type and capacity) for every application, it made sense to put its coding directly in the main program's CPP. But now that we have a templated version, coding changes are no longer necessary, and just like vector and array and deque in the STL, it can be reused "as is" in any program that needs it.

At this point it would make sense to put its coding into a separate file that could be attached to any application's main program CPP, in much the same way that the STL templates are attached. So here's what we will do from now on when we work with templates -- we'll put their code into a "header file" that can be #included into any other CPP. We compare and contrast the way we've been doing it up to now with the way we'll do it going forward:

myProgram.cpp

```
...

class Array
{
  ...
};

int main( )
{
  Array a;
  ...
}

int& Array::operator[](int i)
{
  ...
}

...
```

becomes

Array.h

```
#ifndef Array_h
#define Array_h

template <typename V, int CAP>
class Array
{
  ...
};

template <typename V, int CAP>
int& Array<V, CAP>::operator[](int i)
{
  ...
}

...

#endif
```

myProgram.cpp

```
#include <iostream>
...
using namespace std;

#include <cstdlib>
...

#include "Array.h"

...

int main( )
{
  Array<int, 10> a;
  ...
}
```

The **_h** is added to the templated class' name in the #ifndef and #define statements. It would be nice if the name could match the filename *exactly*, but there cannot be dots in such names -- hence the underscore. Your code editor may actually make up a name for you -- that's okay. It just has to be unique to the project.

The H File

The header file is also called the H file, just like the program file is also called the CPP. *By convention*, the **name of the file** matches the class name exactly, in spelling *and* case, with a dot-h appended. So **class Array** goes in **Array.h**.

For it to be accessible to the main program, a copy of it needs to be in the same folder as the main program's CPP.

It's no longer shown in the sample code, but your identifying comments -- your name and ID and anything else you wish to add -- should appear as the *first two lines* in the H file. There's no place for couts, so no output of identifying information in the H. That goes in the main program's main.

#include

The #include statement in the main CPP essentially does a copy/paste of the contents of the H file into the CPP where it can be compiled. This puts the member function templates *above* main, but that does not matter one bit.

Note the difference between the templated class' #include and the #includes for the C and C++ libraries. The library names are enclosed in angle

brackets, while the H is enclosed in quotes. Angle brackets tell the compiler to look for that file along a folder path that's part of the compiler's installation configuration. Yes, somewhere on you disk are files named "iostream" and "cstdlib", and they are in special folders.

The quotes tell the compiler to look in the working folder -- the one where the CPP is.

#ifndef, #define, and #endif

The content of the H file is contained between a #if and a #endif. That's not C++ code. It's a "compiler directive". It involves a programmer-defined "flag" that can be just about anything, but our convention is for the flag to exactly match the class name in both spelling and case. It needs to be unique to the H file, and this is one good way to assure that. There are other naming conventions -- ones that even your IDE may suggest, and those are fine too, as long as they are unique to the class. Our flag name is **Array**.

#ifndef tells the compiler that if **Array_h** is not already defined as a flag, then compile the code between it and its matching #endif (which is the very last line in any H file). The very first thing that gets done inside the #if container is to define the flag. The #ifndef and the #define have to spell and case the flag name *exactly*, or else it won't work right. The suggestion is to copy and paste the #ifndef line, delete the "ifn" in the copy and type "ine".

Here's what it does -- it allows for multiple #includes of the same H in the same CPP, because after the first time the #ifndef is found, the flag gets set, and all subsequent times will find that the flag *is* defined and skip the code. Why, you may wonder, would we ever do such a thing? Well, we probably would never type it that way. But #includes cascade -- that is, if you #include something that #include's something else, then you get it too. That's why some compilers let you use C++ strings without #including the string library, because something else you #included #includes the string library. More on that later, but suffice it to say that you could have multiple H files that #include each other...

Another Test For Test Driver Programs -- An Updated Checklist

Now that we're writing **#ifndef** containers in our H files, our test drivers should really be testing that their coding is correct. It's called the **"ifndef test"**. Whatever you do, put ifndef tests *in test drivers only* or else it looks... really weird. Here's how an ifndef test is done, using a class named "Array":

Array.h

```
#ifndef Array_h
#define Array_h

template <typename V>
class Array
{
  ...
};

...

#endif
```

Array.**TestDriver**.cpp

```
#include <iostream>
using namespace std;

#include "Array.h"
#include "Array.h" // ifndef test


...

int main( )
{
  ...
}
```

The purpose of the test is to confirm that the #ifndef, #define, and #endif statements are written correctly.

Here are the tests you'll do in any test driver:

1. Test all public functions to make sure they work as you expect them to work.
2. Const object test. (like `const Array c;`)
3. **The "ifndef" test.** *new*

There are two more to be added to this checklist later in this module.

Library Includes

This cascading effect of #includes raises a valid question -- if I use C++ strings in my main CPP, can I just go ahead and **#include<string>** in my H that I also #include, and will that cover me? Well, yes and no. Yes, if you do things in the right order, it will work. But no, because that's not a good

programming practice.

Here's the convention -- if an H file uses something from a C or C++ library, then #include the library just as if it was any ordinary CPP file. A good place to put them if after the #define. If a CPP uses something from a C or C++ library, #include it, and do *ignore* the fact that a #included H file may have already done that for you. If you look at each H and CPP as standing on its own, then you'll avoid compiler problems.

No cout In H Files

The main program should control all input and output, and your data structure H file should do all of its work *independent* of the I/O environment in which it's used. You should be able to use your H file in console apps, network and internet apps, and windowed apps. If you write code in your H that's specific to any I/O environment, you preclude its use in other environments.

What all this means is ***no cout statements in H files***. That usually means no **#include<iostream>** either.