Cmake - An aside

Today we are going to take a break from studying classes and structs to talk about *cmake*. Cmake is a build system which manages the build process in an os and compiler independent manner. You have probably heard of make; you may have cracked a Makefile open or written one yourself. If so, you are probably aware that the syntax can be a bit strange. Cmake is much more straightforward. And we are going to see that presently.

Ok, so first, make sure that you have cmake. If you don't, jump on the interwebs and download the sucker.

Done? Good. If you are running Mac Os (formerly known as OS X), you are going to have to create an alias in your bashrc file, because we are using the terminal. On Mac Os, the actual cmake executable is here:

```
/Applications/CMake.app/Contents/bin/cmake
```

In your .bashrc file, add a line like so:

```
alias cmake="/Applications/CMake.app/Contents/bin/cmake"
```

Open a new shell and type `cmake`. That should work.

If you are on windows, well, find a bash shell (I hear that git ships with one), or hit google, because I don't use windows. I do know that cmake works with windows, but....

Our simple main.cpp file

Ok, lets get going. Create a root project directory. I am calling mine hellocmake. you can call yours whatever.

Cd into the directory and create a main.cpp file. It should look something like the following:

```
#include <iostream>

using std::cout; using std::endl;

int main() {
    cout<< "Hello World...Again...sigh..." << endl;
    return 0;
}
```

Ok. That should have been a big time review. simple. Laughably simple. That isn't the point. We want to compile this, and we want to use cmake to help us, because we are tired of remembering the g++ incantation.

Our first CMakeList.txt file

So lets do this... Cmake uses a file with a very specific name and capitalizaiton to work. So, create a file in the same directory and call it *CMakeLists.txt*. Notice the capitalization; you need to copy this exactly; cmakelists.txt wont do.

Now lets fill it out.

First, we need to set the minimum version for cmake. we do it like so:

```
cmake_minimum_required(VERSION 3.6)
```

Now, I don't know what version of cmake you are using. Actually, we will be able to get away with specifying an earlier version if you only have an earlier version; nothing we are going to do really leverages any new cmake features, but this should be fine assuming you have just downloaded cmake. Otherwise, specify 3.0 or 2.8 or whatever you have.

Next, we need to give our project a name. This doesn't have to be the same name as the parent directory by the way.

```
project(hellocmake)
```

Next we are going to update the compile flags to include the c++ll flag. We are going to use a generic cmake command called *set* to accomplish this.

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
```

This line should make sense if you have done any shell scripting. We are basically assigning the existing value of CMAKE_CXX_FLAGS plus -std=c++11 back to CMAKE_CXX_FLAGS. The argument directly after *set(* is the variable name being set, and the following quoted string is the value being set. Since the value contains a dereference of the variable name (the ${}), as well as a new flag, it takes whatever value it is currently storing and adds the additional flag on the end.

Next, we need to create a new variable to keep track of our source files. (well file, we only have a single file right now).

```
set(SOURCE_CPPS main.cpp)
```

This command works just like before. It takes a space separated list of values. The first value is the variable, and subsequent values are assigned to the variable. If we had more cpp files, we would tack them on to the end.

Ok. Now the last thing we need to do is tell cmake what we want to generate. In this case, we want to create an executable. By the way, we can also generate a static or dynamic library. But for now, lets just create the executable.

```
add_executable( hello ${SOURCE_CPPS} )
```

and that is it. The first parameter to *add_executable* is the executable name. Subsequent parameters are the source file names.

Compiling the executable

Alright, so the first thing we have to do is actually generate a native build file from the cmake file. We do this by calling *cmake*. But we don't really want to call it in the current directory, because cmake creates a lot of temp files. So, we need to create a build directory. So do that. Create a build directory and cd into it.
`mkdir build  cd build`

Yes, I am 100% percent certain that you know how to do this without me creating the previous code block. But those code blocks break up my tedious instructions so...

All right, now, we need to call cmake from this directory but reference the CMakeLists.txt file from the parent directory. Here we go:

```
cmake ..
```

Cmake dot dot. If all goes well, you will have yourself a shiny Makefile, as well as a bunch of other junk. As long as you followed instructions and got things right, you should be golden. If not, go ahead and fix your issues and repeat the instructions. (you may have to delete the directory contents)

I am assuming that you have gotten cmake to work by now. The next step is to type `make`
This should build the executable (whatever you called it). In my case, it is called hello, and it is sitting right here. I can even run it.

So Far

Ok, if you have gotten this far, you are probably a bit peeved. I mean, we just traded a single line

```
g++ main.cpp -o hello
```

for five lines in a CMakeLists file,

```
cmake_minimum_required(VERSION 3.6)

project(hellocmake)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
set(SOURCE_CPPS main.cpp)
add_executable( hello ${SOURCE_CPPS} )
```

and four commands.

```
mkdir build
cd build
cmake ..
make
```

Why would we want to do this? Well, because that simple g++ command gets a whole lot more difficult as we start adding libraries, files, etc. It can quickly get out of hand. Lets add a class into the mix.

Adding a Person Class

Now it is time to take another step into the wide world of C++ development. Up to this point, we have been cheating. All of our work has been going into a single cpp file. In the real world, code is split up between header files, which contain declarations, and cpp files, which contain implementations. We are going to do this grownup style. Create two files:

- Person.h
- Person.cpp

Lets start with the header file (Person.h). The first thing we need to do is add what is called a guard. A guard is a preprocess instruction which prevents the header file from being included multiple times into the same compilation unit during compilation.

TD;LR. When you compile your code, the first step your compiler takes is to expand all of the preprocessor commands. All of the #include directives in each of the cpp files gets replaced by the text from the files they refer to. A compilation unit is basically a cpp file with all of its #includes replaced by their values, along with all of the rest of the preprocessor junk. This expansion is recursive, as the header files you include may well have includes of their own. The guards exist to prevent the preprocessor from copying the same code in multiple times.

Anyhoo, that is a ton of explanation for one line. Sorry.

```
#pragma once
```

Here is the rest of the Person declaration:

```
#include <string>

class Person {
    std::string first_name;
    std::string last_name;
 public:
     Person(const std::string&, const std::string&);
     void greet(const std::string& ) const;
};
```

Pretty simple. Two private variables (first_name, last_name), and two methods - a constructor, and a greet function. Notice anything odd? The method declarations only contain type information. They don't even have parameter names. You can declare them with names, but you don't have to.

Allright. Lets jump over to Person.cpp and actually implement these two functions.

```
#include "Person.h"
#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::string;

Person::Person(const string& fh, const string& ln) :
            first_name(fn),
            last_name(ln) {}

void Person::greet(const string& other) const {
    cout << "Hello " << other << ". My name is " <<
        first_name << " " << last_name << endl;
}
```

Main take aways:

- you need to think of the class name as a namespace and treat it as such once outside of the class declaration. Thus the constructor and the greet function are both prefixed with the class name, followed by double colons. Just like we need to do when addressing things in the std namespace if we don't use the `using` directive.
- speaking of using, it is generally safe to use `using` to get rid of namespaces in the implementation file (cpp). Mainly this is because all of the header files which will be copied into it during compilation will be using fully qualified namespaces. (and you should never use `using` in a header file. got it?)

Updating main

Lets update main to use Person.

```
#include <iostream>
#include "Person.h"

using std::cout; using std::endl;

int main() {
    cout<< "Hello. Bet you didn't think you would still be writing hello world programs." <<
    // who doesn't love alliteration?
    Person p("Jake", "Johanson");
    p.greet("Fred Ferdinand");
```

```
    return 0;
}
```

Updating our CMakeLists.txt file

Ok. Now we need to update our CMakeLists.txt file to be aware of person. First, we add Person.cpp like so:

`set(SOURCE_CPPS main.cpp Person.cpp)`

We also create a *SOURCE_HEADERS* variable

`set(SOURCE_HEADERS Person.hpp)`

And, we go ahead and modify our add_executable like so:

`add_exectuble(hello ${SOURCE_CPPS} ${SOURCE_HEADERS})`

Once that is done, we cd back into our build directory (`cd build`), and remove everything (`rm -rf *`). Then we `cmake .. .` finally, we type make.