Chapter 3 - Cmake - An aside

Today we are going to take a break from studying C++ to talk about *cmake*. Cmake is a build system which manages the build process in an operating system 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, let's get going. Create a root project directory. I am calling mine *hello_cmake*. you can call yours whatever you like.

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;
}
```

Done? That code shouldn't be mysterious to you. It's simple. Laughably simple. But that really isn't the point. We want to compile this, and we want to use cmake to help us do that, because we are tired of remembering the compiler incantation.

Our first CMakeList.txt file

So let's do this... Cmake uses a file with a very specific name and capitalization to do its work. Create a file in the current directory and call it *CMakeLists.txt*. Notice the capitalization; you need to copy this exactly. Cmakelists.txt wont do.

Now let's fill it out.

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

```
cmake_minimum_required(VERSION 3.6)
```

Now, I don't know what version of cmake you are using. Don't just copy the version string above and expect everything to work. Type `cmake --version` and fill out the requirement accordingly. Fortunately, we will be able to get away with specifying an earlier version if that is what you have; nothing we are going to do really leverages any new cmake features, so type in the version of cmake which you have and let us move on.

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)
```

Now that we have given our project a name, we can refer to that name using a cmake variable called *PROJECT_NAME*. In cmake, you reference a variable by wrapping its names in *${}*. To convince ourselves of this, we will learn how to do something else as well - print data to the shell. Cmake has a function called *message* which we will use now to print our project name:

```
message(STATUS "cmake project name: ${PROJECT_NAME}")
```

Next we are going to update the compile flags to include the c++ll flag. To do this, we will use one of the most common cmake commands, called *set*. Set's job is to declare and assign or update variables:

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

The previous 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.

If, for some reason, we had wanted to reset *CMAKE_CXX_FLAGS* to only contain *-std=c++11*, we could have simply typed `set(CMAKE_CXX_FLAGS -std=c++11)`. But we didn't want to so there...

Now that we have seen how to add a compiler flag, and you have read that long winded description, we are going to comment that command out. Cmake uses

hashes to prefix comments:

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

We are going to use a different variable to set the version of C++ we are using:

```
set(CMAKE_CXX_STANDARD 11)
```

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). We will use the *set* command, whose first argument is the name of the variable we wish to set, and whose subsequent, space delimited arguments, comprise the value. Note that we are only setting one value here, but we could set multiple values if desired.

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

Great. 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, let's 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. Notice that we are de-referencing SOURCE_CPPS by surrounding the variable name with *${}* as we did above.

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 in the parent directory. If you are on Linux or Mac, you don't really have to worry about the build system cmake will generate files for; it is going to be Make, plain and simple. If you are on Windows, then Cmake is going to look in the Registry and determine what to generate. 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)

Of course, if you are on windows, or you want to use an alternate generator, you can always get a list of supported generators by running:

```
cmake --help
```

Then, instead of running `cmake ..`, you would run `cmake --G "<NAME OF GENERATOR>" ...`

I am assuming that you have gotten cmake to work by now. The next step is to ask cmake to run your build for you.

```
cmake --build .
```

This should build the executable (whatever you called it). In my case, it is called *hello*, and it is sitting right here in the build directory. 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 (6 if you count the commented out line),

```
cmake_minimum_required(VERSION 3.6)

project(hellocmake)
set(CMAKE_CXX_STANDARD 11)
set(SOURCE_CPPS main.cpp)
add_executable( hello ${SOURCE_CPPS} )
```

and four commands.

```
mkdir build
cd build
cmake ..
cmake --build .
```

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

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 just playing around. All of our work has been

going into a single cpp file. In the real world, code is almost always split up between header files, which contain declarations, and cpp files, which contain implementations. We are going to take this next step grownup style. Create two files:

- Person.h
- Person.cpp

Let's start with the header file (Person.h). The first thing we need to do is add prerocessor directive called a guard. There are two forms which a guard may take. The "classic" form, and the "new" form.

Classic Guard

The classic form consists of three prerocessor directives - two at the top of the file, and one at the end.

The top of the guard looks like this:

```
#ifndef __PERSON_H__
#define __PERSON_H__

// all your code here

#endif
```

New Guard

That is a bit of noise, eh? Most compilers support an alternative guard these days, and it replaces all of that rubbish with a single preprocess directive – `#pragma once`. Just put `#pragma once` at the top of your header file, and you are good to go.

What is a Guard Though?

To understand what a guard is, you have to know a bit more about the compilation process.
When you compile your code, the first step your compiler takes is to execute the preprocessor, which steps through your code and expands the preprocessor directives, creating **compilation units** out of each of your cpp files.

A **compilation unit** is basically a file which has had all of its `#include` directives replaced by the contents of the files which they point to, along with all of the rest of the preprocessor commands replaced / expanded in place. This expansion is recursive, as the header files you include may well have `#include`s of their own.

This preprocessor is not particularly smart, however. It doesn't really know anything about c++ semantics. It doesn't even keep track of the files which it has already processed. So, the guards exist to prevent the preprocessor from copying the same code into a compilation unit multiple times, and potentially recursing forever.

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

```
#pragma once
```

Here is the rest of the Person declaration. We will delve into classes later, but in the meantime copy this verbatim. For now, the cliff notes are as follows: A method whose declaration bears the same name as the class, and doesn't return anything is a constructor, which is basically like `__init__` in Python, except that you can define multiple constructors in C++ with different signatures. Any variable declared after `public:` may be accessed by users of the class using dot notation, just like in Python. Anyway, we are really looking at cmake here so copy the following:

```
#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.

All right. Let's jump over to Person.cpp and actually implement these two functions. Notice that inside the cpp file, we have to namespace each method with the name of the class. That is what the `Person::` business is about.

```
#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

Let's 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)
```

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

```
cmake ..
```

Finally, we type:

```
cmake --build .
```

From now on, we are going to be using **cmake** to do our bidding.