C++ Ecosystem – Part 1

References

- https://cmake.org/
- https://www.johnlamp.net/cmake-tutorial.html
- https://riptutorial.com/cmake
- https://clang.llvm.org/docs/ClangFormat.html
- https://clang.llvm.org/docs/ClangFormatStyleOptions.html
- https://devblogs.microsoft.com/cppblog/clangformat-support-in-visual-studio-2017-15-7-preview-1/
- https://github.com/google/googletest
- https://github.com/isocpp/CppCoreGuidelines
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Installing WSL for Windows

- https://docs.microsoft.com/en-us/windows/wsl/install-win10
- https://www.windowscentral.com/how-install-wsl2-windows-10

VM for macOS (to install Linux, so you can install g++ more easily)

- https://mac.getutm.app/

Building the latest gcc on WSL

- https://solarianprogrammer.com/2017/05/04/building-gcc-wsl-windows-subsystem-linux/
- Then create a symlink: ln -s *path*/to/file *path*/to/symlink
- Better
    - https://stackoverflow.com/questions/50729550/installing-g-on-windows-subsystem-for-linux (use 9 instead of 7)
    - Then make symlinks in usr/bin to gcc and g++
- Maybe even better
    - https://linuxize.com/post/how-to-install-gcc-compiler-on-ubuntu-18-04/

Preface

C++ The ecosystem surrounding the C++ language is vast, impossible to discuss in any one particular place, however an overview of some useful elements is warranted as part of these notes. Each development team selects from this ecosystem in support of their project, no two are probably the same. As you work with different teams, you'll be exposed to different tools and practices. In order to help prepare you for what you may see in the future, three tools and one topics are covered in this series. These four items are *CMake*, a meta-project build system, *Clang-Format* a tool used to automatically format code, *Google Test* a unit testing framework, and finally something known as the *C++ Core Guidelines*, from the designer of C++, which provides direction on writing simpler, more efficient, and more maintainable code.

CMake

All of the code samples in these notes are developed as cross-platform C++ code, up to and include the C++ 17 standard. Additionally, the samples utilize a cross-platform build system, or meta-project build system, called CMake. From the CMake web site (https://make.org)…

> "CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner. Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. Simple configuration files placed in each source directory (called CMakeLists.txt files) are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC) which are used in the usual way. CMake can generate a native build environment that will compile source code, create libraries, generate wrappers and build executables in arbitrary combinations. "

In other words, CMake is used to generate a platform specific build system. On Windows, an MS Visual Studio solution, among others, can be generated and then Visual Studio used for the development environment. On Linux, a Makefile can be generated and then used to build the code. On both platforms an Eclipse project generated and then Eclipse used as the development environment. There are many other possibilities, but this gives you an idea of what can be done.

The purpose of this section is to familiarize you with enough information about CMake to understand how to generate native project files from the provided samples and create basic CMake configuration files for your own projects. For those interested in a deeper understanding of the syntax of the CMake configuration files, please visit the CMake web site at https//:cmake.org.

CMake Tools

CMake comes in two parts, a command line utility that handles the task of creating the platform specific project files, and a GUI that provides a nice user interface for the building of the platform specific project files.

Before using CMake, it needs to be installed on your computer. For most users, it is enough to download and install from the installers provided on the CMake website. Alternatively, for Linux systems such as Ubuntu, using a package installer like *apt* is recommended. When using a package installer it might be necessary to install both ***cmake*** and ***cmake-qt-gui***.

Make sure the version of CMake you install is at least CMake 3.12 or greater, due to the use of how the C++ 20 standard is specified.

CMake Hello World

CMake configuration files have the name `CMakeLists.txt` and contain a declarative style language for defining how a C++ project should be built. A simple C++ project with C++ source files contained within a single folder will have a single `CMakeLists.txt` file. A more sophisticated project, with multiple folders and possibly multiple build targets might have many different `CMakeLists.txt` files located in different folders, with each `CMakeLists.txt` file specifying build details for a specific project component. For now, this introduction will utilize a single `CMakeLists.txt` configuration file. Later on, as larger projects are built, additional information about preparing multiple `CMakeLists.txt` configuration files is presented.

Referring back to the Hello World program discussed in the introduction to C++, we can now examine the `CMakeLists.txt` configuration file. The parenthesis and numbers in the parenthesis are not part of the text in the file, they are there to identify statements in the file.

```
(1) cmake_minimum_required(VERSION 3.12)
(2) project(HelloWorld)
```

```
(3) add_executable(HelloWorld main.cpp)

(4) set_property(TARGET HelloWorld PROPERTY CXX_STANDARD 20)
```

Each line is detailed next.

1. This line specifies the minimum version of CMake that may be used. In the case of the samples in these notes, CMake version 3.12 and greater must be used in order to correctly handle the way the C++ 20 standard is specified.
2. The `project` command gives a name for the project, this is required. The authors of CMake recommend placing this command near the top of the configuration file, but after setting the CMake minimum version.
3. The `add_executable` command specifies an executable target. A single C++ project may have any number of different executable targets, with different source files being associated with them.
    - The first parameter is the name to give to the executable. Do not put .exe or any other extension here. If necessary, based on the generated platform, an appropriate extension is specified during project generation.
    - After the name, is a list of all the source files to associate with the executable target. In this case, there is only one, but if there were more, each should be placed inside the add_executable statement separated by a space.
4. The `set_property` command, in this case, is used to specify the C++ 20 standard for the compiler. Based on this setting, the generated project build file(s) will instruct the compiler to compile using the C++ 20 standard.

I have used all lowercase for the CMake commands, but this is not a requirement of CMake. Any combination of upper and lowercase is allowed; CMake commands are not case sensitive.

Generating Project Files

With the `CMakeLists.txt` file completed, it is now possible to use the CMake utilities to generate a platform specific build project. This section details organizing and generating platform specific project build files.

When using CMake it is important to keep the source files and project files in separate folders. If you are used to something like Visual Studio where the project and source files are located in the same folder tree structure, you'll want to learn to follow a different pattern when using CMake; it may take time, but you'll eventually like it. To keep the source files and generated projects files separate, create a `build` folder (the folder can be any name) that is a sub-folder of the main project folder and instruct CMake to generate the platform specific build files in that location. When CMake generates the build environment it generates project file links back to the source folder.

The following steps detail generating the platform specific project files for the ***Hello World*** code sample. These steps use the CMake GUI application, but the project files can also be generated using the command line utility if desired.

1. Create a `build` folder that is a sub-folder to the `Hello World` folder.
2. Start the CMake GUI application.
3. Select the ***Browse Source***… button.
4. Navigate to the `Hello World` folder.
5. Select the ***Browse Build***… button.
6. Navigate to the `build` folder location.
7. Press the ***Configure*** button.
8. Choose the desired generator options. On Windows I have made the following selections:

- ***Specify the generator for this project***: Visual Studio 17 2022
- ***Optional platform for generator***: x64
- ***Optional toolset to use***: left empty
- Radio button for ***Use default native compilers*** selected

9. After specifying the generator options, press the ***Finish*** button.
    - I find that the first time after configuration, it is necessary to press ***Configure*** again to cause the red highlighted “errors” to resolve.
10. Press the ***Generate*** button.

At this point, the platform specific project files are generated and located in the `build` folder. You can now navigate to that folder and utilize the build environment.

Enabling Compiler Options

- https://foonathan.net/blog/2018/10/17/cmake-warnings.html
- https://arne-mertz.de/2018/07/cmake-properties-options/

It is considered a best practice to enable a high-level of compiler warnings in order to help identify potential bugs in code and eliminate all compiler warnings before code it accepted into the product. Some consider it a good idea to have the compiler fail if there are any warnings!

Because our goal is to compile correctly across at least two C++ compilers, Microsoft and g++, we need a way to specify compiler specific options; because the compilers don’t support all of the same options. There are a number of ways to do this using CMake, this section details only one method, the use of target_compile_options, along with the use of CMake conditionals.

At project generation time, the CMake script needs to know which build platform it is generating for, thankfully, there are CMake variables defined to help with this. CMake defines a CMAKE_<LANG>_COMPILER_ID string that identifies the compiler vendor. For Microsoft Visual C++ it is `MSVC`, for the GNU Compiler Collection it is `GNU`.

CMake allows for conditionals through the use of an if-elseif-else-endif structure. Combined with testing for the compiler, it looks like…

```
if (CMAKE_CXX_COMPILER_ID STREQUAL "MSVC")
    ...do MSVC specific stuff...
elseif (CMAKE_CXX_COMPILER_ID STREQUAL "GNU")
    ...do g++ specific stuff...
elseif (CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    ...do llvm specific stuff...
endif()
```

The value of a variable in CMake is accessed by surrounding it with `{}`. To convert that value to a string, surround it with double quotes `""`. To compare two strings in a conditional, use the `STREQUAL` operator.

The warnings options to enable for the Microsoft C++ compiler are:

- `/W4` https://docs.microsoft.com/en-us/cpp/build/reference/compiler-option-warning-level?view=vs-2019
- `/permissive-` https://docs.microsoft.com/en-us/cpp/build/reference/permissive-standards-conformance?view=vs-2019

The warnings options to enable for the g++ compiler are:

- `-Wall` https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc/Warning-Options.html#Warning-Options
- `-Wextra` https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc/Warning-Options.html#Warning-Options
- `-pedantic` https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc/Warning-Options.html#Warning-Options

Using `-Wall` with the Microsoft C++ compiler works, but generates a lot of warnings from the standard library header files (yes, the question is, "why do they?"). Because all code is being compiled by both the Microsoft and g++ compilers, any warnings generated by `-Wall` are handled by the g++ compiler.

The `target_compile_options` function in CMake allows a specific compile target to be specified with options. It is possible to set options for the global project, but that may not be desired, depending on the nature of the project (e.g., multiple languages), therefore we use `target_compile_options`. The first parameter to the function is the CMake target, which in the case of the `HelloWorld` example is, `HelloWord` (specified in the `add_executable` statement). With this knowledge, here is how to set the compiler warnings for each compiler vendor:

```
if (CMAKE_CXX_COMPILER_ID STREQUAL "MSVC")
    target_compile_options(HelloWorld PRIVATE /W4 /permissive-)
elseif (CMAKE_CXX_COMPILER_ID STREQUAL "GNU")
    target_compile_options(HelloWorld PRIVATE -Wall -Wextra -pedantic)
elseif (CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    target_compile_options(HelloWorld PRIVATE -Wall -Wextra -pedantic)
endif()
```

Multiple Source Files

Most applications have more than a single source file in them, therefore, we need a way to specify those source files as belonging to a particular executable build target. It turns out to be as simple as continuing to add each of these files to the `add_executable` command, as shown next:

```
add_executable(BigProject
	main.cpp
	utilities.hpp
	utilities.cpp
	simulation.hpp
	simulation.cpp)
```

When creating a Visual Studio project, for example, the generated project will create appropriate `Header Files` and `Source Files` project sections and organize accordingly.

Another approach to specifying multiple files is to create variables that have the names of the files and then use those variables in the `add_executable` command. Consider the following CMake commands:

```
set(HEADER_FILES
	utilities.hpp
	simulation.hpp)

set(SOURCE_FILES
	main.cpp
	utilities.cpp
	simulation.cpp)

add_executable(BigProject ${HEADER_FILES} ${SOURCE_FILES})
```

Using the set command, we specify the name of a variable and then the value(s) to store. In the first `set` command, the `.hpp` files are stored in the `HEADER_FILES` variable. The second `set` command stores the `.cpp` filenames in the `SOURCE_FILES` variable. Finally, these variables are referenced in the `add_executable` command. To use the values stored in a variable the `${}` syntax is used, around the name of the variable.

There isn't a requirement that `.hpp` and `.cpp` files need to be separated in this manner, I've done it as a way to organize files by a logical grouping in my mind, it doesn't affect the project files generated by CMake.

Formatting Code with ClangFormat

ClangFormat is a utility that formats source code (C++ in our case) based upon customizable settings. ClangFormat is one part of the LLVM compiler infrastructure (http://llvm.org/). There are many ways to use the utility, this section discusses three approaches to its utilization. The first is to manually run the utility from the command line, the second is to integrate into Visual Studio, and the third is to integrate it into the build chain, using CMake.

Reference: https://clang.llvm.org/docs/ClangFormat.html

Style Customization

ClangFormat provides a lot of flexibility in how it will format code, based on various style settings. These settings can be specified as command line parameters or in a configuration file. These notes are based around the placement of the formatting options stored in configuration file. The list of formatting options available are found at this link: https://clang.llvm.org/docs/ClangFormatStyleOptions.html

The format of config is YAML (YAML Ain't Markup Language), which is basically a `key : value` pair organization. Comments are indicated by the `#` character followed by the comment, on a single line.

The name of the configuration file can be either `.clang-format` or `_clang-format`. I personally recommend using `_clang-format` to ensure the file is (typically) visible while navigating the file system. To have clang-format utilize the configuration file add the `-style=file` option to the command line. It confuses many first time (and even experienced) users, but `file` is not a placeholder for the name of the file, it is a literal. The command line option is literally `-style=file`. With this option specified, clang-format will look for either of `.clang-format` or `_clang-format`.

If you want to know the default ClangFormat settings for your system, you can use the `-dump-config` command line option to report the settings to the console. Alternatively, this output can be redirected to a file. For example:

```
clang-format -dump-config > config.txt
```

If any of the defaults are desired to be changed, they can be overridden in the configuration file. In fact, you might consider dumping the default config to a file and then modifying it to match the project needs and then give it the filename of either `.clang-format` or `_clang-format`.

For any option you want to change, add the name of the option (the key), followed by a colon and then the value for that option. The following shows an example `_clang-format` configuration file.

```
Language: Cpp
BreakBeforeBraces : Allman
IndentCaseLabels: true
NamespaceIndentation: Inner
IndentWidth: 4
TabWidth: 4
UseTab: Never
DerivePointerAlignment: false
PointerAlignment: Left
```

The first line specifies the language for the following settings. A single configuration file can specify settings for multiple languages, as may be needed for a multi-language project.

A project only needs a single style file, usually placed in the root project folder. If the `clang-format` utility doesn't file a style file in the current directory, it navigates through the parent folders until it finds one (if it exists) and uses it.

The next three sub-sections assume the latest ClangFormat (Clang 9 at the time of this writing, but version 8 is probably okay for this course) is installed and the binaries folder specified in the environment path.

Command Line Usage

The simplest use is to run the utility from the command line. The name of the executable is clang-format. The following link details the command line options for the utility: https://clang.llvm.org/docs/ClangFormat.html

Anytime a file is desired to be formatted, it can be done from the command line. Let's consider the following command line usage:

```
clang-format -i -style=file main.cpp
```

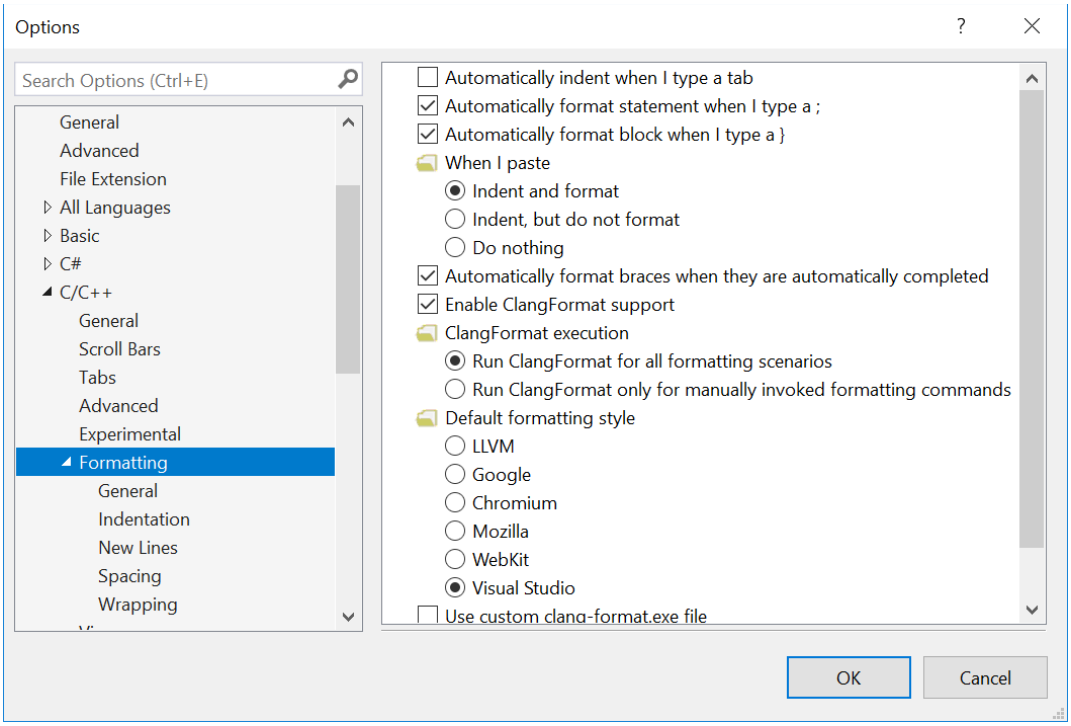
A closer look at each item in this command:

- `clang-format` : This is the executable name of the utility. On windows you may need to add a .exe extension; but not necessary in PowerShell.
- `-i` : Perform an in-place modification of the source file. By default, `clang-format` sends the formatted code to standard out (the console). That is useful if you don't want to modify the original source file, but that isn't what we are doing. We want the original file to be modified.
- `-style=file` : This instructs the utility to use a configuration file to override default settings.
- `main.cpp` : The name of the source file to format. This example shows only a single file, but any number of files can be on the command line.

While we won't be resorting to using ClangFormat from the command line, it is useful to understand these options as that knowledge is necessary for the next two subjects, integration in Visual Studio and integration into the build chain with CMake.

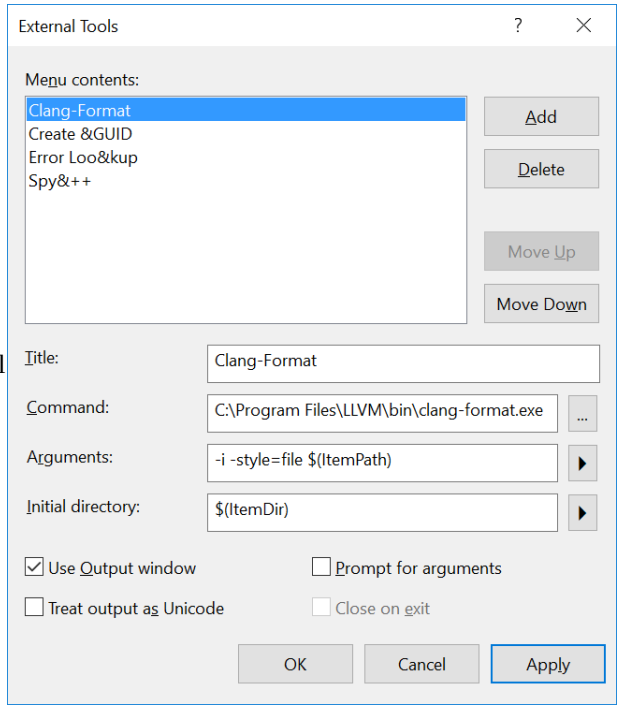
Visual Studio Integration

Visual Studio 2017 and later have automatic detection and use of a ClangFormat configuration file. There is a Visual Studio extension available, but it only works with the Professional edition of Visual Studio. The screenshot to the right shows where to find the code formatting options in Visual Studio.



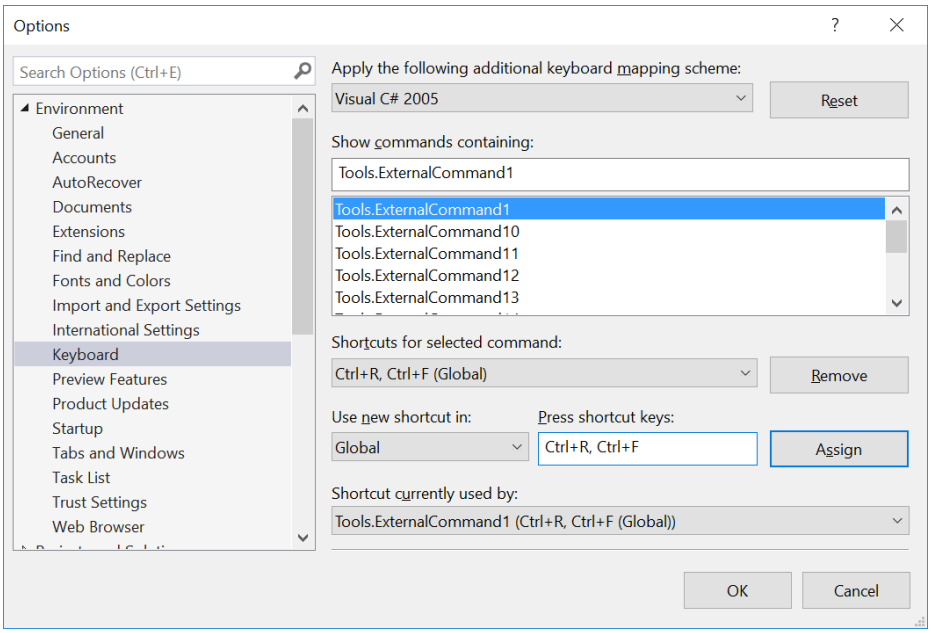
In the case neither of these are available or possible with your Visual Studio installation, there is another way to integrate clang-format with Visual Studio that provides the main desired benefit...the ability to use a keyboard shortcut to format the current source file. This is accomplished by using the ***External Tools*** option. The following step-by-step procedure details setting this up; this procedure assumes clang-format is already installed on your computer.

1. From the ***Tools*** option of the main menu, select ***External Tools…*** This brings up a dialog where you can specify command line and parameters for (unsurprisingly) external tools.
2. Press the ***Add*** button and then set the options to match the settings in the image to the right.
    1. The location of `clang-format.exe` may be in a different folder on your computer. Be sure to navigate and select the correct location for your computer.
    2. Be sure to check the ***Use Output window*** checkbox. This will direct output from the command to the output window inside of Visual Studio.
3. When finished with the settings, select the ***Apply*** button.
4. Use the ***Move Up*** button to place the new Clang-Format external tool as the first item in the list. This is necessary for the way we are going to setup a keyboard shortcut to execute this tool.
5. When finished, press the ***OK*** button to close the dialog.



The set of steps detail preparing a keyboard shortcut to execute the newly added Clang-Format external tool.

1. From the ***Tools*** option of the main menu, select ***Options…*** This brings up a dialog where numerous IDE settings can be customized. We are interested in setting up a keyboard shortcut to invoke the new Clang-Format external tool we just added.
2. From this dialog, open the ***Environment*** section and then select ***Keyboard***.
3. In the section of the dialog labeled ***Show command containing:*** type in ***Tools.ExternalCommand1***. You will see the options below this label refine. From this list, select ***Tools.ExternalCommand1***.
4. Move the cursor to the box labeled ***Press shortcut keys:*** You can press any shortcut keys you like, but I recommend ***Ctrl-R, Ctrl-F***. This is the combination used by the clang-format extension, that's why I recommend it.
5. When complete, the dialog should look like the one to the right.
6. When done, press ***OK*** to close the dialog.



In Visual Studio, with an implementation or header file open (.cpp/.hpp), press the ***Ctrl-R, Ctrl-F*** combination to have the file formatted with clang-format. Remember, clang-format is running from the command line. Therefore, any unsaved edits are not captured by clang-format. Be sure to save the source file before invoking clang-format on it.

CMake Pre-Build Step Integration

In the previous two approaches, it is up to the developer to decide when/if ClangFormat is applied. Meaning that incorrectly formatted code can make it into the project repository. A best integration for ClangFormat is to have it automatically applied, ensuring all source code is always formatted. The use of something like a git (GitHub) hook would solve this by applying code formatting anytime new code is added to the repository, that is out of the scope of this class. However, we can come close to that by have all code automatically formatted during the build processes. It would still be possible for a malicious developer to change code after building, but that is probably someone you won't want on your team in the first place. This rest of this section discusses adding a custom step in the build process to apply ClangFormat using CMake.

In order to run `clang-format` automatically over all files in a project, we need to do the following tasks:

1. Define a variable or variables to hold the names of the source files in the project.
2. Find the location of the `clang-format` utility.
3. Add a custom build target that runs `clang-format`.
4. Set the custom build target as a dependency of the primary project target.

For these steps, I'll be extending the ***HelloWorld*** project to add a ClangFormat pre-build step.

Step 1 – Files Variable

The section on CMake has already discussed using a variable to hold the names of the source files. There is only a single source file, resulting in a simple variable specification:

```
set(SOURCE_FILES
    main.cpp)
```

Step 2 – Find clang-format

We need the generated project to know the location of the clang-format utility. CMake has a command called `find_program` that helps do this. This command is used to search the folders found in PATH environment variable, along with other folders that are specified as options. As long as the path to the `clang-format` utility is in the PATH environment variable, `find_program` does what we need. The following command demonstrates this:

```
find_program(CLANG_FORMAT "clang-format")
```

The first parameter is the name of a variable to place the full pathname of the found program (if found). The second parameter is the name of the executable to find. This command is run when the native platform project files are generated. If the program is found, `CLANG_FORMAT` has the location, otherwise it remains undefined. Because of that, an if statement can be used to conditionally add the custom target, as shown in the next step.

Step 3 – Create Custom Target

There are two parts to putting together the custom target. The first is to obtain the full file system pathname of each source file, for use in the clang-format command. The second is creating the custom target with the clang-format command.

The first step looks like this:

```
unset(SOURCE_FILES_PATHS)
foreach(SOURCE_FILE ${SOURCE_FILES})
       get_source_file_property(WHERE ${SOURCE_FILE} LOCATION)
       set(SOURCE_FILES_PATHS ${SOURCE_FILES_PATHS} ${WHERE})
endforeach()
```

The `unset` command ensures the variable `SOURCE_FILES_PATHS` has no value. The `foreach` loop then goes through each of the files defined in `SOURCE_FILES` and obtains the full pathname, adding each to the `SOURCE_FILES_PATHS` variable. When complete, `SOURCE_FILES_PATHS` contains all the source files with their full pathname.

The second step is to create the custom target, with the `clang-format` command to format all the project source files. It looks like this:

```
add_custom_target(
        ClangFormat
        COMMAND ${CLANG_FORMAT}
        -i
        -style=file
        ${SOURCE_FILES_PATHS})
```

The `add_custom_target` command creates another sub-project in the native build project, with the name ***ClangFormat***. In the case of Visual Studio, a project named ***ClangFormat*** is created and added to the ***HelloWorld*** solution.

Following `COMMAND` is the name of the program to execute, which in this case is the `clang-format` program. After the name of the program to run are any command line parameters. The parameters should follow the same pattern as running the program from the command line.

Step 4 – Set Project Dependency

In order for the ClangFormat custom target to run every time the main ***HelloWorld*** project is build, it needs to be set as a dependency of ***HelloWorld***. This is done using the `add_dependencies` command, as shown below:

```
add_dependencies(HelloWorld ClangFormat)
```

With this set, anytime the ***HelloWorld*** project is built, the ***ClangFormat*** project is first executed, ensuring all source files are correctly formatted.

When complete. the full `CMakeLists.txt` file looks looks like…

```
cmake_minimum_required(VERSION 3.12)
project(HelloWorld)

set(SOURCE_FILES
    main.cpp)

add_executable(HelloWorld ${SOURCE_FILES})
set_property(TARGET HelloWorld PROPERTY CXX_STANDARD 20)

find_program(CLANG_FORMAT "clang-format")
if (CLANG_FORMAT)
        unset(SOURCE_FILES_PATHS)
        foreach(SOURCE_FILE ${SOURCE_FILES})
                get_source_file_property(WHERE ${SOURCE_FILE} LOCATION)
                set(SOURCE_FILES_PATHS ${SOURCE_FILES_PATHS} ${WHERE})
        endforeach()

        add_custom_target(
                ClangFormat
                COMMAND ${CLANG_FORMAT}
                -i
                -style=file
                ${SOURCE_FILES_PATHS})

    add_dependencies(HelloWorld ClangFormat)
endif()
```

Unit Testing with Google Test

Google Test, or *GTest*, is a unit testing framework for C++ projects. It provides easy to use facilities for doing testing on functions or classes. The testing framework is compiled along with the tests and the code being tested. Similar to the integration of ClangFormat, we'll use CMake to integrate GTest into our project, while also creating a separate build/executable target for running tests, leaving the main project target alone.

Most of the information in this section is documented in more detail on the Google Test github site. It is located at: https://github.com/google/googletest Follow the link to the Google Test Primer.

Integrating with CMake

One of the most interesting things (to me) that can be done with CMake is to have it download (via git) and integrate external code into a project. Two different techniques for doing this are shown below. Both techniques are shown in these notes because depending on the code you work with, you might see either or both techniques used and it is important to be familiar with both.

In this sub-section we are going to define a CMake configuration file as before, defining our project, setting up clang-format, and then provide instructions for downloading and integrating Google Test into our project. In the end, we'll have a native build system with a build/executable target that is the project without any testing code, and another build/executable target that executes all of the unit tests.

Current Technique

Reference: https://google.github.io/googletest/quickstart-cmake.html

Starting with CMake version 3.14 a command called `FetchContent` is available; it is actually a family of commands. This command can be used to download and integrate third-party code into another project. The command is complex, allowing for a lot of flexibility in how it is used; recommend reading the documentation for full details. The approach taken here is to utilize CMake in combination with Git to clone and integrate Google Test. To do this, the following options are specified in the `FetchContent_Declare` command:

- `GIT_REPOSITORY` This is set to the git URL of the source
- `GIT_TAG` This is set to the git branch or tag to checkout

For Google Test the final result looks like:

```
include(FetchContent)
FetchContent_Declare(
    googletest
    GIT_REPOSITORY    https://github.com/google/googletest.git
    GIT_TAG           v1.15.0
)
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googleTest)

target_link_libraries(UnitTestRunner gtest_main)
```

In order to utilize this file in the main project's `CMakeLists.txt` file, several steps are necessary.

1. Organize the variables used to hold the names of the source files.

   ```
   set(HEADER_FILES utilities.hpp)
   set(SOURCE_FILES utilities.cpp)
   set(UNIT_TEST_FILES TestUtilities.cpp)
   ```

2. Create two executable targets, one for the main project, and the other for the unit test execution. There is a main.cpp file with a main function in it for the main project. Additionally, there is a `TestUtilities.cpp` file with a `main` function it in that executes the unit tests.

   ```
   add_executable(GoogleTestIntro ${HEADER_FILES} ${SOURCE_FILES} main.cpp)
   add_executable(UnitTestRunner  ${HEADER_FILES} ${SOURCE_FILES} ${UNIT_TEST_FILES})
   ```

3. Update the clang-format section to capture all the project files

   ```
   foreach(SOURCE_FILE ${HEADER_FILES} ${SOURCE_FILES} ${UNIT_TEST_FILES} main.cpp)
   ```

4. Use the `set` command to disable a compiler setting that Google Test likes to set. The set command instructs Google Test to use dynamically linked shared libraries (shown above).

5. Use the `target_link_libraries` command to add the Google Test shared libraries to the test runner target defined earlier in the CMake configuration file (shown above).

** Show the full CMakeLists.txt (GoogleTestIntro project) file at this point **

Older Technique

Reference: https://stackoverflow.com/questions/38006584/how-to-clone-and-integrate-external-from-git-cmake-project-into-local-one

CMake provides a powerful command that facilitates the downloading, patching, configuration, building, and installation of, well, external code. The command is `ExternalProject_Add`. The command is complex, allowing for a lot of flexibility in how it is used; recommend reading the documentation for full details. The approach taken here is to utilize CMake in combination with Git to clone and integrate Google Test. To do this, the following options are specified in the `ExternalProject_Add` command:

- `GIT_REPOSITORY` This is set to the git URL of the source
- `GIT_TAG` This is set to the git branch or tag (or even commit hash) to checkout
- `SOURCE_DIR` This is set to the location, on the local file system, for where to place the cloned project
- `BINARY_DIR` This is set to the location, on the local file system, for where to build the cloned project

For Google Test the final result looks like:

```
ExternalProject_Add(googletest
  GIT_REPOSITORY    https://github.com/google/googletest.git
  GIT_TAG           master
  SOURCE_DIR        "${CMAKE_CURRENT_BINARY_DIR}/googletest-src"
  BINARY_DIR        "${CMAKE_CURRENT_BINARY_DIR}/googletest-build"
  CONFIGURE_COMMAND ""
  BUILD_COMMAND     ""
  INSTALL_COMMAND   ""
  TEST_COMMAND      ""
)
```

There are some other settings that need to be set to empty strings, those are the last four in the above example.

To facilitate modularity, we'll place this command into its own CMake configuration file and include it in our primary `CMakeLists.txt` file. The way this is done is to create a `CMakeLists.txt.in` file. This file follows the same format and language as `CMakeLists.txt`, but is intended to be included in other CMake configuration files. In the case of Google Test, the full `CMakeLists.txt.in` file is:

```
cmake_minimum_required(VERSION 3.12)

project(googletest-download NONE)

include(ExternalProject)

ExternalProject_Add(googletest
  GIT_REPOSITORY    https://github.com/google/googletest.git
  GIT_TAG           master
  SOURCE_DIR        "${CMAKE_CURRENT_BINARY_DIR}/googletest-src"
  BINARY_DIR        "${CMAKE_CURRENT_BINARY_DIR}/googletest-build"
  CONFIGURE_COMMAND ""
  BUILD_COMMAND     ""
  INSTALL_COMMAND   ""
  TEST_COMMAND      ""
)
```

In order to utilize this file in the main project's `CMakeLists.txt` file, several steps are necessary.

6. Organize the variables used to hold the names of the source files.

```
set(HEADER_FILES utilities.hpp)
```

```
set(SOURCE_FILES utilities.cpp)
set(UNIT_TEST_FILES TestUtilities.cpp)
```

7. Create two executable targets, one for the main project, and the other for the unit test execution. There is a main.cpp file with a main function in it for the main project. Additionally, there is a `TestUtilities.cpp` file with a `main` function it in that executes the unit tests.

```
add_executable(GoogleTestIntro ${HEADER_FILES} ${SOURCE_FILES} main.cpp)
add_executable(UnitTestRunner  ${HEADER_FILES} ${SOURCE_FILES} ${UNIT_TEST_FILES})
```

8. Update the clang-format section to capture all the project files

```
foreach(SOURCE_FILE ${HEADER_FILES} ${SOURCE_FILES} ${UNIT_TEST_FILES} main.cpp)
```

9. Use the `configure_file` command to update the configuration to match the system it is being used on. This command takes as its first argument an input file and copies it to another location, while substituting variable references with the appropriate value. In the example above `${CMAKE_CURRENT_BINARY_DIR}` is substituted in the output file.

10. Use the `execute_process` command to create a child process to execute the git command.

11. Use the `execute_process` command to create a child process to generate the native platform build files.

12. Use the `set` command to disable a compiler setting that Google Test likes to set. The set command instructs Google Test to use dynamically linked shared libraries.

13. Use the `add_subdirectory` command to create a build target for gtest.

14. Use the `target_link_libraries` command to add the Google Test shared libraries to the test runner target defined earlier in the CMake configuration file.

** Show the full CMakeLists.txt (GoogleTestIntro project) file at this point **

Terminology

Let's start by reviewing some terminology before moving on to the use of Google Test. These terms and definitions come from the Google Test documentation.

- ***Test*** Uses one or more assertions to verify the tested code's behavior. If a test crashes or has failed assertions, it fails, otherwise it succeeds.
- ***Test Suite*** Composed of one or more tests. Tests should be grouped into test suites that reflect the structure of the tested code.
- ***Test Program*** The program used to execute the tests (test suites). A test program may contain more than one test suite.

Testing Assertions

Google Test provides a number of macros that are used to make assertions about some code's behavior. Typically the assertions take an expected value and an actual value and compare them for equality, greater than, less than, among others. The following are examples of what the assertions look like:

```
ASSERT_EQ(computed, 8);    // assert 'computed' contains the value 8
ASSERT_LE(computed, 8);    // assert 'computed' is less than 8

EXPECT_EQ(computed, 8);    // assert 'computed' contains the value 8
EXPECT_LE(computed, 8);    // assert 'computed' is less than 8
```

The difference between `ASSERT_*` and `EXPECT_*` is that the failure of an assertion with `ASSERT_*` causes the remainder of the assertions in that test to not be run; Google Test documentation specifically says "abort the function." It is recommended to use `EXPECT_*` to show all assertion failures in the test function, unless there is a meaningful reason to abort the test function after the first assertion failure.

The `ASSERT_*` and `EXPECT_*` macros are useful for most of the data types you need, however, they aren't always adequate for testing floating point numbers. Sometimes a small amount of expected difference, or error, is tolerable. Therefore, Google Test provides assertion macros to facilitate this.

The first set of macros look like:

```
ASSERT_FLOAT_EQ(computed, expected);
EXPECT_FLOAT_EQ(computed, expected);
```

These macros succeed if the two values are close to each other. Close means the values differ by no more than four Units in the Last Place (ULP). Strongly recommend reading the Google Test documentation and link it refers to for a complete understanding.

There are another set of macros that look like:

```
ASSERT_NEAR(computed, expected, absError);
EXPECT_NEAR(computed, expected, absError);
```

These macros succeed if the `computed` value is within the +- value of `expected` by `absError`.

Writing Simple Tests

Google Test includes a `TEST` macro that provides the ability to define a test function as part of a test suite. Use of the `TEST` macro looks like a function, and inside of it any valid C++ code can be written. Key to this code are assertions, these assertions are used to indicate the success or failure of the test(s) defined in each `TEST` macro. Let's say we want to test the `swap` function from earlier in these notes:

```
void swap(int& x, int& y)
{
	int temp = x;
	x = y;
	y = temp;
}
```

We want to write a test that verifies this function works correctly. We can define a simple test for it like this:

```
TEST(SwapTests, SwapTest)
{
    int a = 1;
    int b = 2;
    cs3460::swap(a, b);
    EXPECT_EQ(a, 2);
    EXPECT_EQ(b, 1);
}
```

The first parameter is the name of a test suite. The test suite does not have to be pre-defined anywhere, any declaring it is part of a test suite are correctly recognized by Google Test. The second parameter is the name for the specific test defined by the `TEST` macro. The body of the `TEST` macro is the testing code, including the assertions necessary to validate the code.

Let's take a look at the `sin` function, also from earlier in these notes:

```
double sin(double angle)
{
	return (4 * angle * (180 - angle)) / (40500 - angle * (180 - angle));
}
```

Testing of this function requires more than a single, simple assertion. Instead, we want to test some well known angles, along with a series of other angles. The following code shows the tests for the `sin` function.

```
TEST(SinTests, SinSimple)
{
    EXPECT_EQ(cs3460::sin(0), 0.0);
    EXPECT_EQ(cs3460::sin(30), 0.5);
    EXPECT_EQ(cs3460::sin(90), 1.0);
    EXPECT_EQ(cs3460::sin(150), 0.5);
}

TEST(SinTests, SinAll)
{
    const auto SIN_TOLERANCE = 0.002;

    std::array angle = {
        0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0,
        100.0, 110.0, 120.0, 130.0, 140.0, 150.0, 160.0, 170.0};
    std::array result = {
        0.0,
        0.17364817766693,
        0.342020143325669,
        0.5,
        ... // removed some values for brevity here
        0.5,
        0.342020143325669,
```

```
            0.17364817766693
        };

        for (auto index = 0; index < angle.size(); index++)
        {
            EXPECT_NEAR(cs3460::sin(angle[index]), result[index], SIN_TOLERANCE) <<
    "expected " << result[index] << " actual " << cs3460::sin(angle[index]);
        }
    }
```

This time there are two `TEST` macros used, each having different tests, but both belonging to the same `SinTests` test suite. The `SinSimple` test compares the results against some exact expected values. The `SinAll` test uses the floating point "near" assertions to validate a wider range of values.

Another interesting thing this code demonstrates is the ability to customize an error message by streaming it to an assertion.

Using Test Fixtures

** Still to write **

Executing Tests

It is a simple matter to run the tests defined in a project. In the main function of the project, use the following two statements:

```
testing::InitGoogleTest(&argc, argv);
return RUN_ALL_TESTS();
```

For detailed information on what is taking place with these two statements and to understand additional customization, refer to the AdvancedGuide; link available from the Google Test Primer documentation.

Anytime the executable with these two statements is executed, the unit tests are run, without output directed to the console.

Core CPP Guidelines

Clang-Tidy

http://clang.llvm.org/extra/clang-tidy/

Conan

https://conan.io/

Installing g++ 14 & CMake

Use the following instructions to get g++ 14 installed on a Debian (e.g., Ubuntu) based linux system.

1. `sudo apt update`
2. `sudo apt upgrade` (may take a while)
3. ~~`sudo add-apt-repository 'deb http://mirrors.kernel.org/ubuntu noble main universe'`~~
4. ~~`sudo apt install build-essential`~~ ~~(brings in a number of possibly needed build utilities)~~
5. ~~`sudo add-apt-repository ppa:ubuntu-toolchain-r/test`~~
6. ~~`sudo apt update`~~
7. `sudo apt install gcc-14 g++-14`
8. `sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-14 110 --slave /usr/bin/g++ g++ /usr/bin/g++-14 --slave /usr/bin/gcov gcov /usr/bin/gcov-14`
9. `g++ --version`
    - If this shows 14.0 or higher, go to step 9
    - If this shows less than 14.0, may need to do
        - `sudo update-alternatives --config gcc`
9. ~~`sudo add-apt-repository -r 'deb http://mirrors.kernel.org/ubuntu noble main universe'`~~ ~~(this removes the repository, just as a cautionary measure)~~
10. ~~`sudo apt update`~~
11. `cd`
12. `mkdir cmake`
13. `cd cmake`
14. `sudo apt remove cmake` (may want to log out and back in after running this)
15. `wget https://github.com/Kitware/CMake/releases/download/v3.30.2/cmake-3.30.2-linux-x86_64.sh`
16. `chmod +x ./cmake-3.30.2-linux-x86_64.sh`
17. `./cmake-3.30.2-linux-x86_64.sh`
    1. space bar when you see –more--
    2. select ‘y’ when it asks about the license
    3. select ‘n’ when it asks about the sub-directory
18. `cd ..`
19. Edit the file `.bashrc`
    1. At the very end of the file add: `PATH=$PATH:~/cmake/bin`
20. `source ~/.bashrc` (to force the path to refresh, alternatively log out and log back in)

Installing Clang-Format

Windows

1. Visit the LLVM web page, located at llvm.org.
2. From this page, on the right-hand side, select the “available for download” link.
3. In the section labeled “Download”, select the latest release, the one labeled “via Git”.
4. In the section entitled, “Pre-Built Binaries”, select the “Windows (64-bit)” option.
5. Once it finishes downloading, run the installer.
    - Select the option to “Add LLVM to the system PATH for current user”, or the other one for all users if you like.

This actually installs the whole clang suite of tools, including the clang C++ compiler.

Linux

1. ~~Visit the LLVM web page, located at llvm.org.~~
2. ~~From this page, on the right-hand side, select the “available for download” link.~~
3. ~~In the section labeled “Download”, select the latest release, the one labeled “via Git”.~~
    - ~~You may need to directly visit this link instead: https://releases.llvm.org/download.html#12.0.0~~
4. ~~In the section entitled, “Pre-Built Binaries”, follow the link to the GitHub release page.~~
5. ~~Copy the link to the appropriate pre-build binaries. For my WSL Ubuntu installation I copied this link: https://github.com/llvm/llvm-project/releases/download/llvmorg-12.0.0/clang+llvm-12.0.0-x86_64-linux-gnu-ubuntu-20.04.tar.xz~~
6. Open a bash shell.
7. Use wget to download the file
    - `wget https://github.com/llvm/llvm-project/releases/download/llvmorg-17.0.2/clang+llvm-17.0.2-x86_64-linux-gnu-ubuntu-22.04.tar.xz`
8. Extract this archive using: `tar -xf clang+llvm-17.0.2-x86_64-linux-gnu-ubuntu-22.04.tar.xz`
9. Edit the file `.bashrc`
    1. At the very end of the file add: `PATH=$PATH:~/clang+llvm-17.0.2-x86_64-linux-gnu-ubuntu-22.04/bin`
10. `source ~/.bashrc` (to force the path to refresh, alternatively log out and log back in).
11. Run `clang-format --version` to verify the installation.

This actually installs the whole clang suite of tools, including the clang C++ compiler.