Week 8

Exercise 68

First, the files that are to be compiled should be collected into a container of some sort. The assumption is, to keep it simple, that all `.cc` files in or under the current working directory should be compiled, and thereafter linked. This can be accomplished using the `fs::recursive_directory_iterator` function, which searches a folder and its subfolders for any files, returning path iterators that can then be converted to strings for convenience.
Another possibility is for the command line argument to specify the folder to compile the files therein. This argument should then be removed from the array, ensuring that any modifiers that are specified thereafter (i.e. `-pthread`) are the only elements left in the array.
Of course, only the relevant files should be kept (i.e. the `.cc` files). A simple `remove_if` algorithm solves this.
Second, the relevant files should be compiled. A queue-like structure could be implemented here, but another possibility is to simply successively create vectors of threads, sized to the number that is specified by the user (which is, again, thereafter deleted from the `argv` array), and then waiting until these threads have finished before starting the next group of threads. Specifically, `pipe` and `popen` can be used to execute a system command and catch its output. Here, it is especially important to catch any error (specifically run-time errors) that may result from using these commands. Now, there are two options. First, as the exercise specifies, any time there is any output at all, that would mean there has occurred some kind of error or warning, which should be displayed, and the loop of creating successive threads should halt. Second, and I think preferably, compilation could simply continue, outputting any kind of error that other files may exhibit, which would allow the user to correct more errors than simply the first one(s). Of course, the step thereafter, linking, is pointless if there are any errors and should be skipped entirely.
If, however, a class structure is implemented, this is how it could look:

Class	**Collector**	**Compilers**
Data	**d_Folder (string)** Specifies folder to search for .cc files	**d_vFilepathsSub (vector)** Contains strings representing a subset of .cc files
	d_vFilepaths (vector) Contains strings representing .cc files	**d_mErrors (map)** Stores the output from the compile commands
Functions	**iterateFolder** Finds the files in the specified folder	**compile** Compiles the files in d_vFilepathsSub and catches any output (esp. run-time errors, but also g++ errors)
	findCC Deletes any files from d_vFilepaths that are not .cc files	

Once a specified client (i.e. `Compiler`) is done, then it acquires a mutex and displays its d_mErrors to cerr, with associated filenames.
The actual command used to compile the `.cc` files is as folows:

```
string const command = "g++ -fdiagnostics-color=always --std=c++17 -Wall -O2 -c -o ./tmp/o/117"
+ filename + string(".o ") + filepath;
```

N.B.: The `-fdiagnostics-color=always` switch forces the colors that `g++` errors typically output to also be present in this case.
The same `pipe` and `popen` process is then used to link the files together. Considering the fact that all `.cc` files were (ideally) compiled, the following command can be used to link the resulting `.o` files in a subfolder of the current working directory:

```
g++ -o ./tmp/bin/binary ./tmp/o/*.o
```

N.B.: That folder is chosen to ensure some compatibility with Icmake. If, however, another folder was chosen instead of the current working directory, the `tmp` folder of that location is used here instead.
Of course, since sometimes static libraries must be specified (e.g. in the case of using the filesystem or pthread libraries), this is where any optional argv arguments should be appended.
Lastly, the folder containing the temporary files is deleted (i.e. `./tmp/bin/o`).