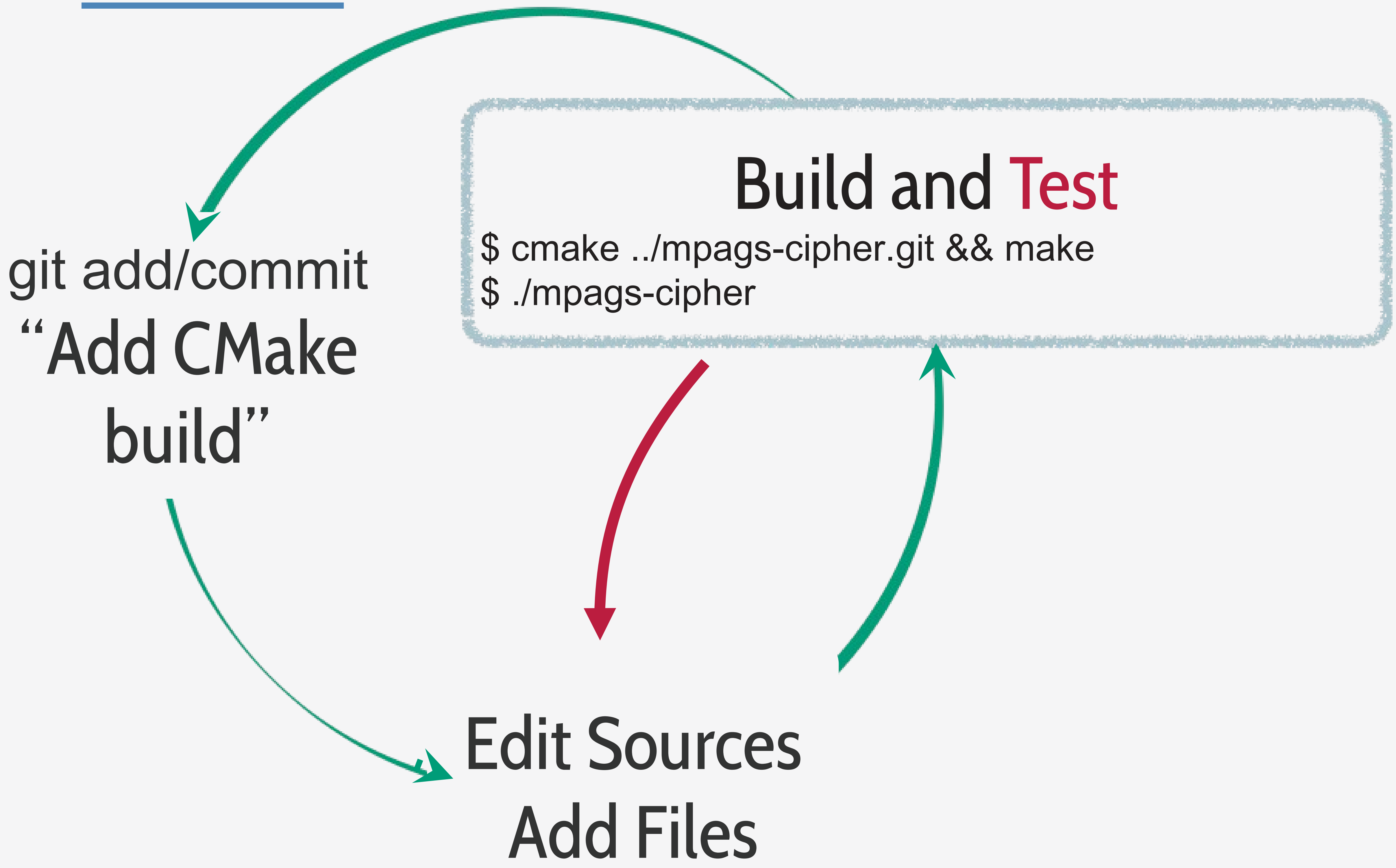
Unit Testing mpags-cipher with Catch and CMake

- Mark Slater (based on slides from Ben Morgan)

THE UNIVERSITY OF WARWICK

UNIVERSITY OF BIRMINGHAM

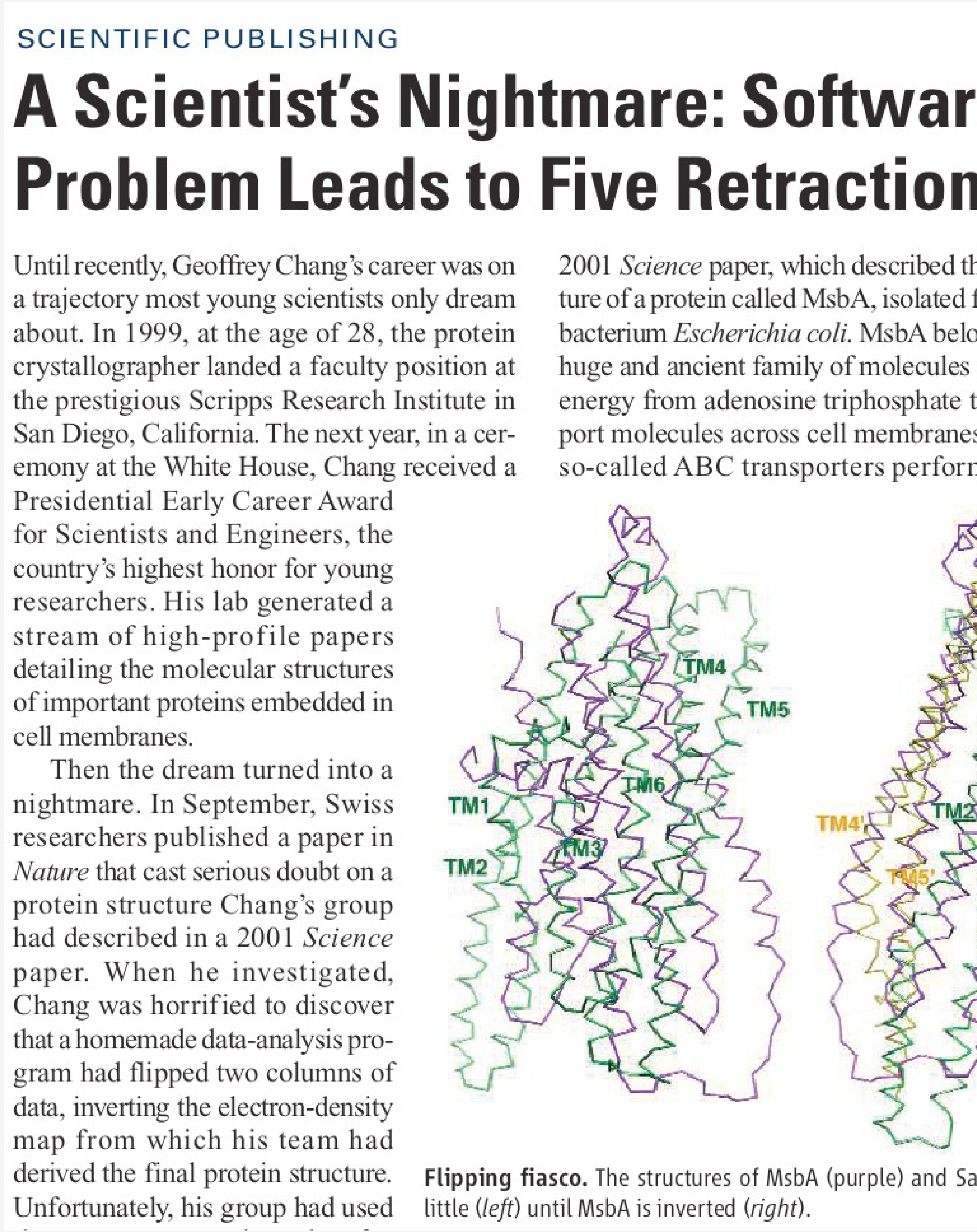
Developer Workflow



```
$ cmake ../mpags-cipher.git && make
$ ./mpags-cipher
```

Why Test At All?

- *"I'm a scientist, show me a plot, I'll know if it's right or not"*
- **How** do you know?
- The code changed, then the plot changed - is it still right?
    - Again, how do you know?
- If you know, then by definition there's a metric to measure "rightness", and thus something an (unbiased) computer can measure!



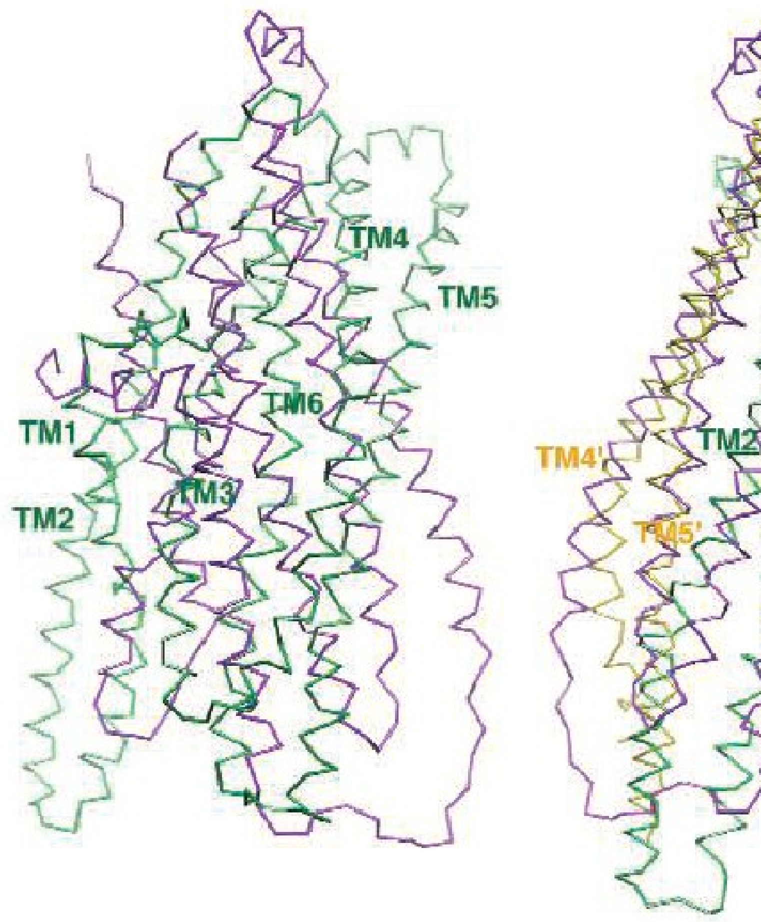
SCIENTIFIC PUBLISHING

A Scientist's Nightmare: Softwar
Problem Leads to Five Retraction

Until recently, Geoffrey Chang's career was on a trajectory most young scientists only dream about. In 1999, at the age of 28, the protein crystallographer landed a faculty position at the prestigious Scripps Research Institute in San Diego, California. The next year, in a ceremony at the White House, Chang received a Presidential Early Career Award for Scientists and Engineers, the country's highest honor for young researchers. His lab generated a stream of high-profile papers detailing the molecular structures of important proteins embedded in cell membranes.

Then the dream turned into a nightmare. In September, Swiss researchers published a paper in *Nature* that cast serious doubt on a protein structure Chang's group had described in a 2001 *Science* paper. When he investigated, Chang was horrified to discover that a homemade data-analysis program had flipped two columns of data, inverting the electron-density map from which his team had derived the final protein structure. Unfortunately, his group had used

2001 *Science* paper, which described th
ture of a protein called MsbA, isolated f
bacterium *Escherichia coli*. MsbA belo
huge and ancient family of molecules
energy from adenosine triphosphate t
port molecules across cell membranes
so-called ABC transporters perforn



Flipping fiasco. The structures of MsbA (purple) and Sa
little (*left*) until MsbA is inverted (*right*).

Unit Testing

- We now have several “units” in mpags-cipher: command line parsing, input preprocessing and the Caesar Cipher.

- **Unit Testing simply means writing a small program that exercises a given “unit” by providing a series of known inputs and checking the outputs are as required for that input. The tests pass, i.e. the program runs successfully, if the outputs are as required.**

- Whilst we can write these programs ourselves, it’s more usual to use a **Unit Testing Framework** that provides functions and objects specialised for this task. This allows us to concentrate on the contents of the tests.

- Another reason is to ensure the tests themselves are correct!

Catch.hpp

- We've chosen the Catch unit testing framework for this course purely for simplicity
  - Others include gtest, Boost, CPPUnit
- It comes as a single header which we've supplied for you under the Testing subdirectory of mpags-cipher
- See its GitHub page for further info and documentation:

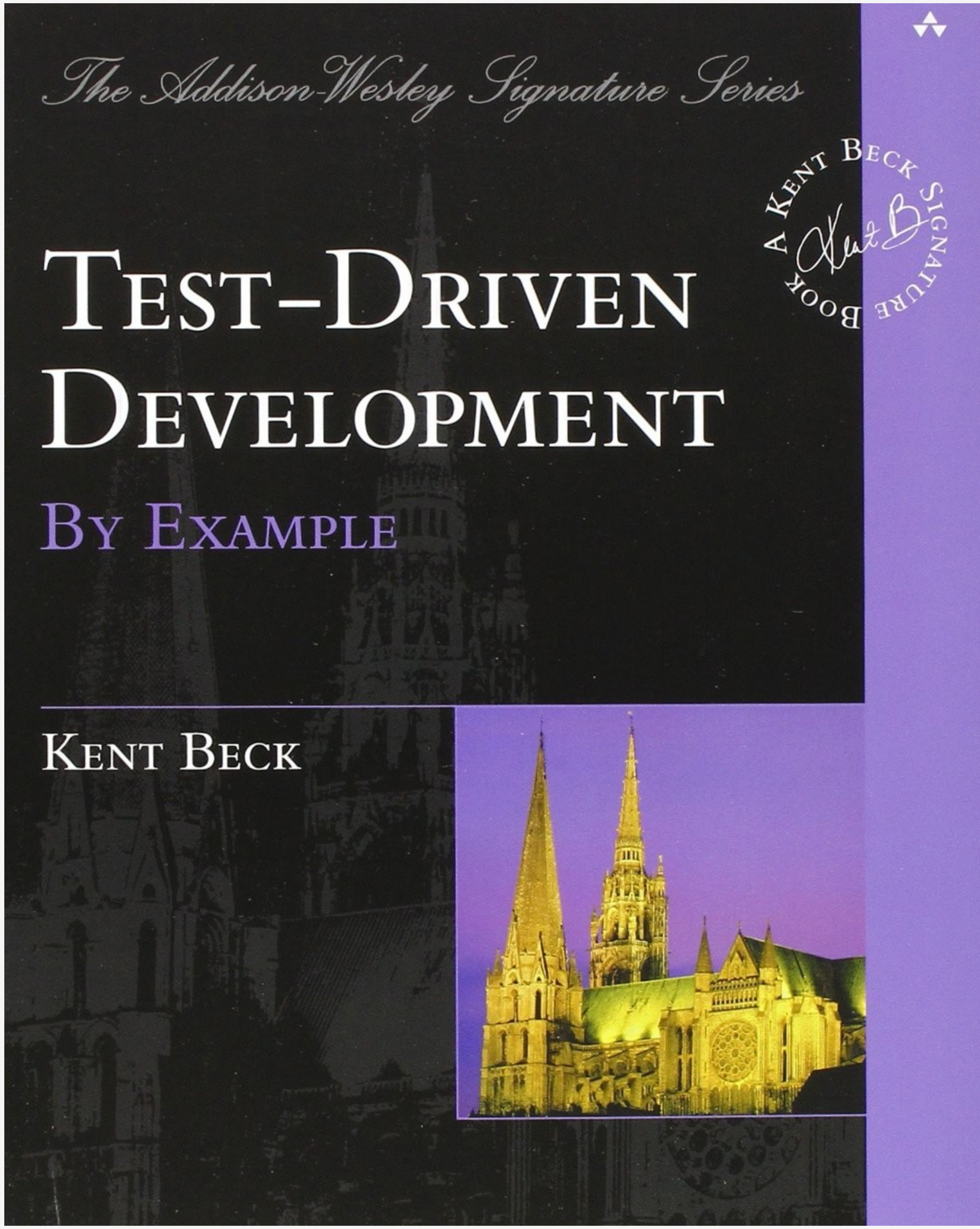
https://github.com/catchorg/Catch2

Unit Testing Regressions

- Tests can also help us to fix bugs and to quickly spot if they reoccur.
- Imagine a user of your software reports a problem - the ciphertext they are getting is not as expected when they encrypt “helloworld”
- To help identify and resolve the issue, you write a test that reproduces the bug to provide a starting point. Other tests may be written as you diagnose and resolve the bug.
- This “bug test” is kept and run as part of testing in the future in case further changes cause it to reappear (i.e. cause a **regression**)

Testing Resources

- Naturally a huge topic and not C++ specific.
- A good and compact starting point is the Kent Beck book on the right
- Though the examples are in Java, the process and ideas are applicable to C++ and other languages
- The [Addison-Wesley Signature Series]() provides many other useful titles on testing topics


The Addison-Wesley Signature Series
A Kent Beck Signature Book
Test-Driven Development
By Example
Kent Beck

Walkthrough: Testing mpags-cipher with Catch and CMake

- In the following walkthrough, we'll prepare mpags-cipher for unit testing and write the first few tests for it.

- We'll start by splitting the build of mpags-cipher into a **library** of functions that is linked to the actual mpags-cipher executable. This will allow us to test the functions easily without multiple recompilations.

- With the library in place, we'll use CMake and its CTest system to add a very basic test program. We'll see how building and running the test integrates with our workflow

- Finally, we'll use Catch to write our first true unit test and see how to build and run it under CMake/CTest.

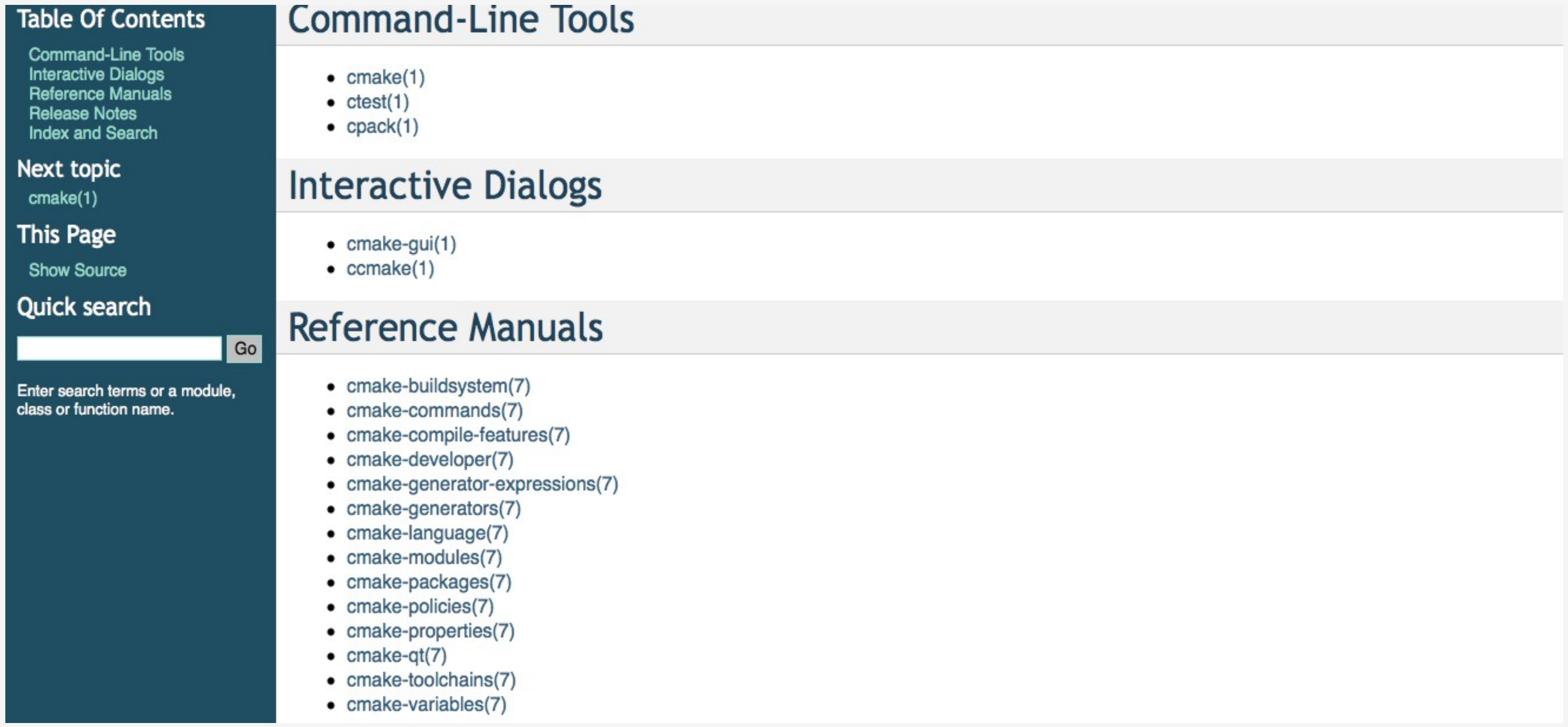


Table Of Contents

- Command-Line Tools
- Interactive Dialogs
- Reference Manuals
- Release Notes
- Index and Search

Next topic

cmake(1)

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

Command-Line Tools

- cmake(1)
- ctest(1)
- cpack(1)

Interactive Dialogs

- cmake-gui(1)
- ccmake(1)

Reference Manuals

- cmake-buildsystem(7)
- cmake-commands(7)
- cmake-compile-features(7)
- cmake-developer(7)
- cmake-generator-expressions(7)
- cmake-generators(7)
- cmake-language(7)
- cmake-modules(7)
- cmake-packages(7)
- cmake-policies(7)
- cmake-properties(7)
- cmake-qt(7)
- cmake-toolchains(7)
- cmake-variables(7)

docker

CMake

Catch2

Tools you’ll need

How to Test mpags-cipher?

- Though we have units we'd like to test in mpags-cipher, they are all compiled into a monolithic executable, so we can't test them independently and in isolation.

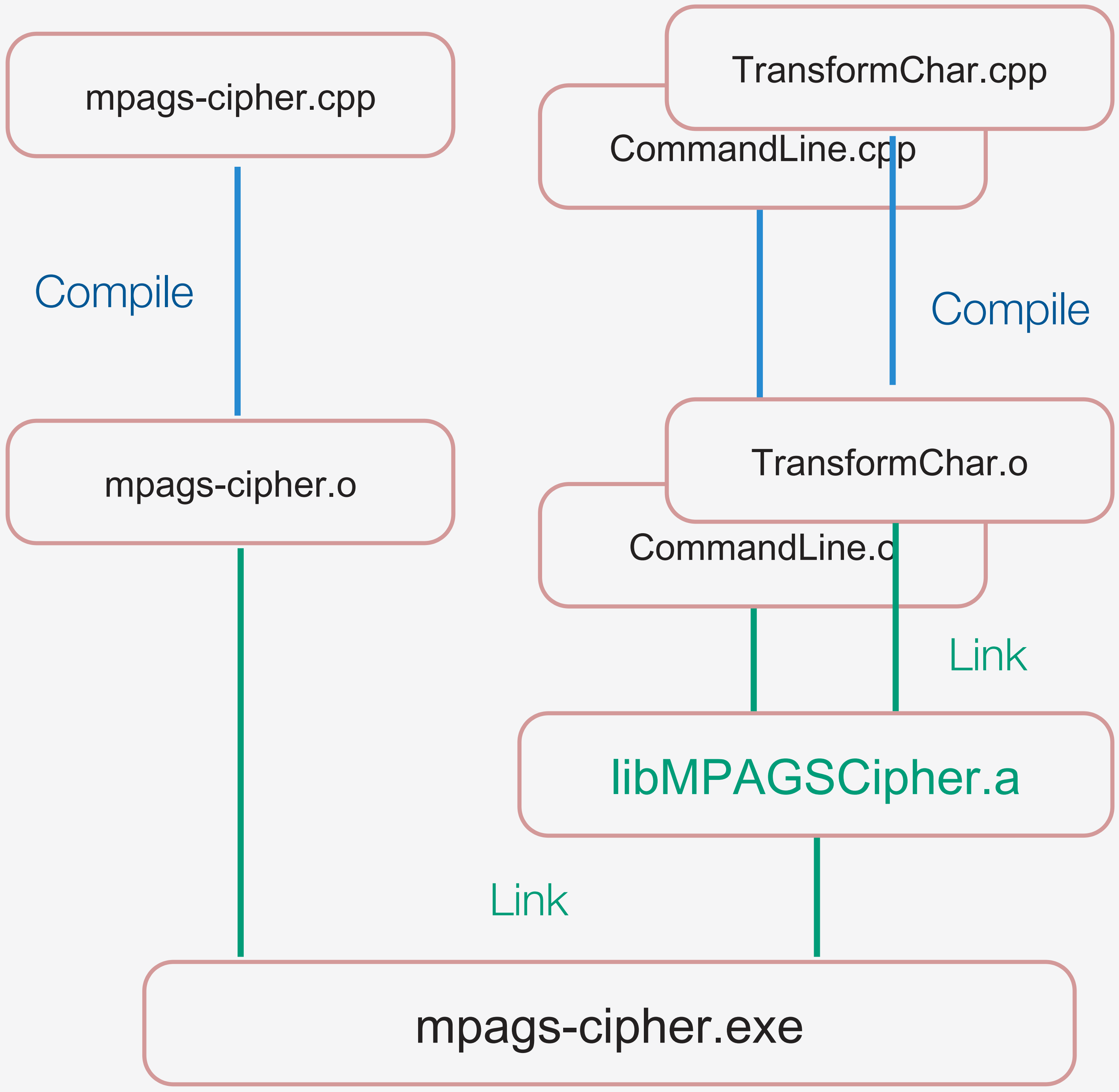
- We could build our test programs like we do for mpags-cipher, creating an executable composed of the test code plus the unit of code, e.g. TransformChar.cpp, we want to test.

- However, the unit may use other units, so we'd need to compile those and know that we need to, plus we'd be be recompiling the same code for each executable it is used in.

- **Instead, we're going to bundle the units into a ready compiled block of binary code that many executables can reuse - a Library.**

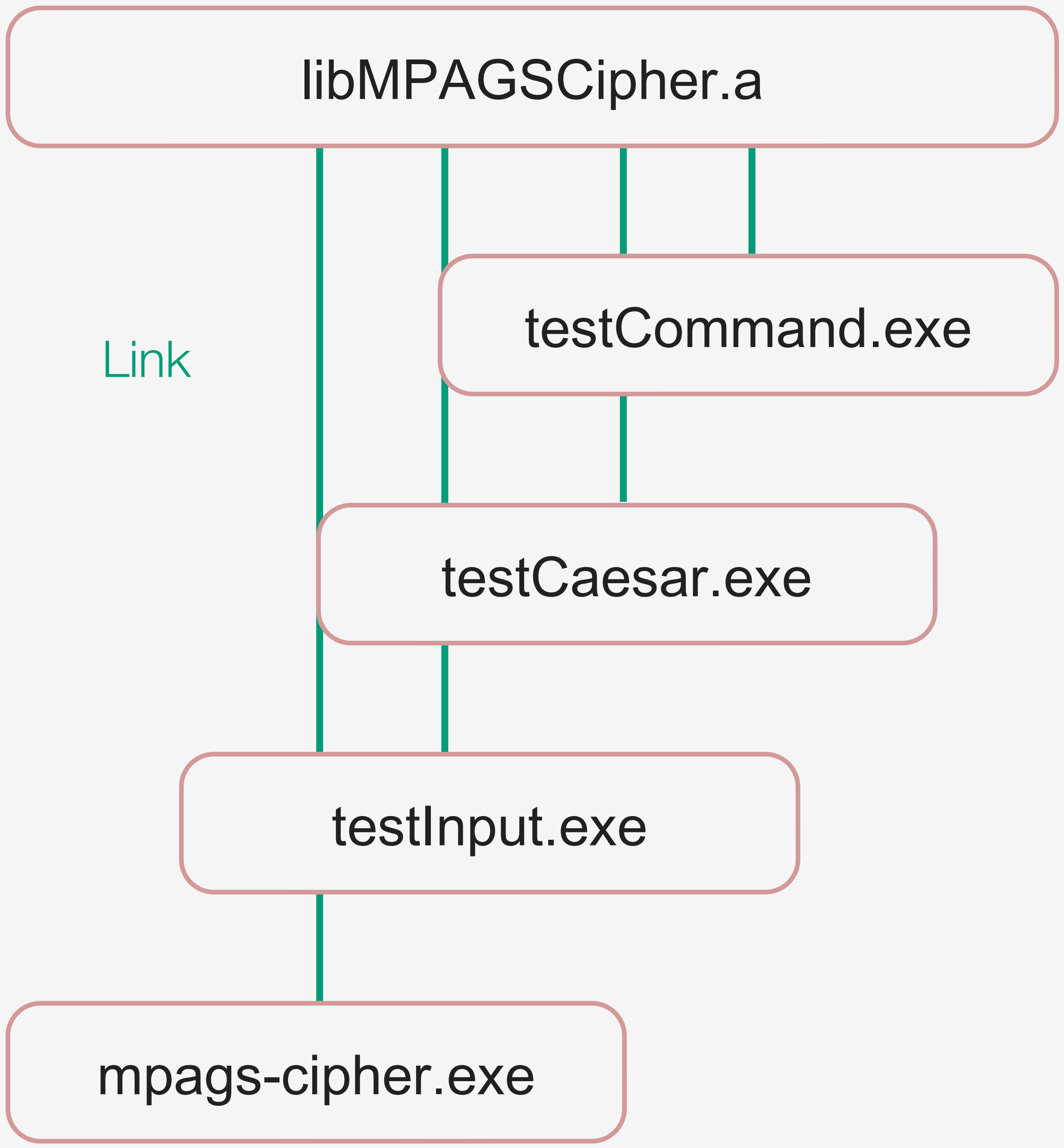
Libraries in C++

- Have already seen how an executable is compiled and linked from multiple sources.
- A Library is just an intermediate, but persistent, step that bundles compiled object files into a special file - the library itself.
- A Library can be linked to an executable (and even other libraries) just as object files are.



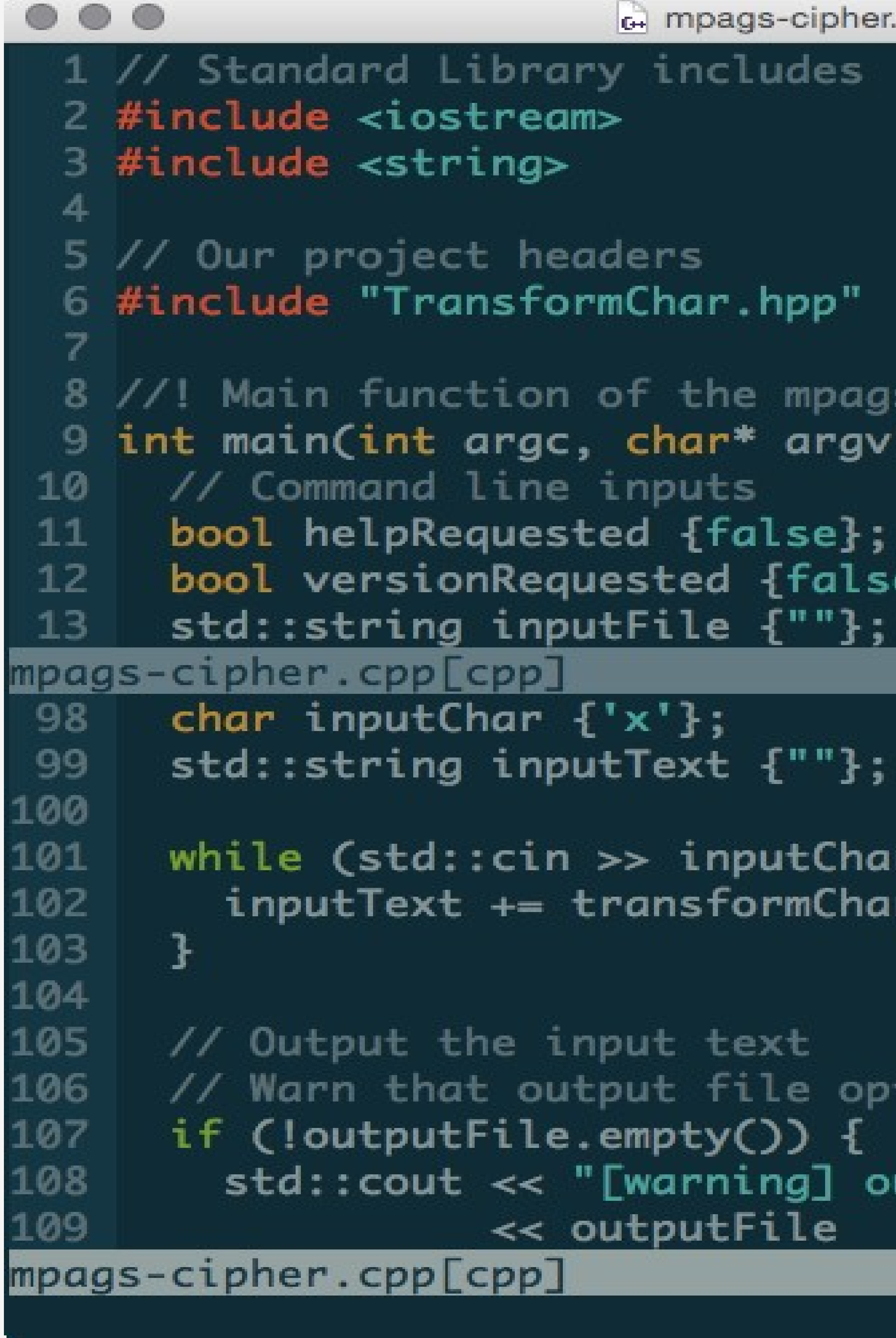
Advantages

- We can have as many executables as we want linking to the library - as needed for testing!
- Each executable uses the same library code, so the code only needs to be compiled once rather than individually every executable.
- Apart from timesaving, this also reduces the potential for errors caused by compile differences



Using a Library

- No real difference to compiling all the code together:
- We #include headers from the library declaring the interfaces we want to use
- Use of interfaces is identical
- **However, must link our executable to the library to ensure it can use the binary implementation of the interfaces we've used.**

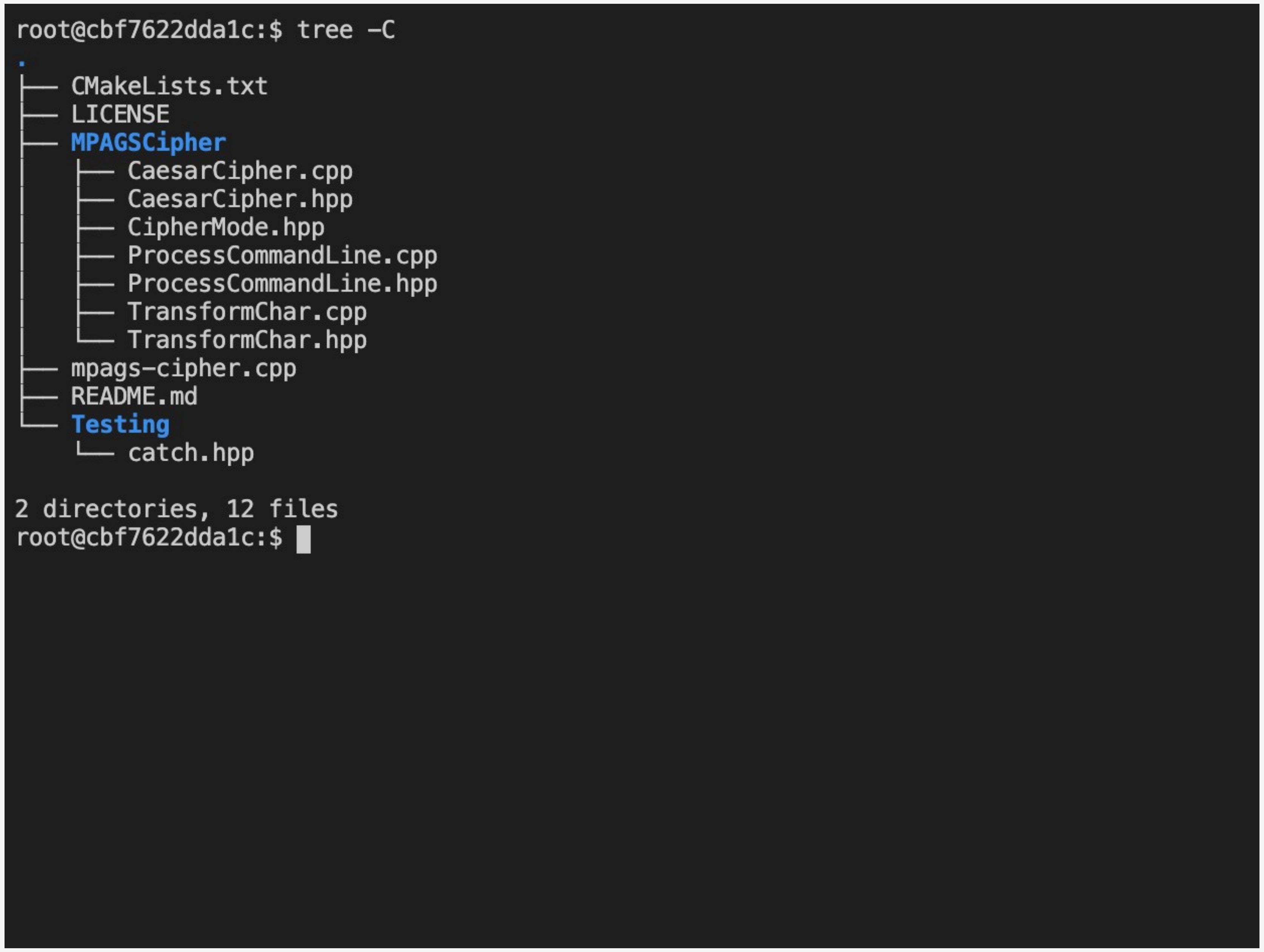

mpags-cipher.

```
// Standard Library includes
#include <iostream>
#include <string>

// Our project headers
#include "TransformChar.hpp"

//! Main function of the mpag
int main(int argc, char* argv
  // Command line inputs
  bool helpRequested {false};
  bool versionRequested {fals
  std::string inputFile {""};
mpags-cipher.cpp[cpp]
  char inputChar {'x'};
  std::string inputText {""};

  while (std::cin >> inputCha
    inputText += transformCha
  }

  // Output the input text
  // Warn that output file op
  if (!outputFile.empty()) {
    std::cout << "[warning] o
              << outputFile
mpags-cipher.cpp[cpp]
```

1: Project Structure for Libraries

The project structure may now be becoming a bit clearer - the code that implements the actual functionality of mpags-cipher is provided as a series of headers/sources under MPAGSCipher/. We'll use CMake to compile this code into a library and link it to the mpags-cipher program.

Whilst we've only used a single CMake script, we'll now see how to split up the build into a top level script plus one for building the library



```
root@cbf7622dda1c:$ tree -C
.
├── CMakeLists.txt
├── LICENSE
├── MPAGSCipher
│   ├── CaesarCipher.cpp
│   ├── CaesarCipher.hpp
│   ├── CipherMode.hpp
│   ├── ProcessCommandLine.cpp
│   ├── ProcessCommandLine.hpp
│   ├── TransformChar.cpp
│   └── TransformChar.hpp
├── mpags-cipher.cpp
├── README.md
└── Testing
    └── catch.hpp

2 directories, 12 files
root@cbf7622dda1c:$
```

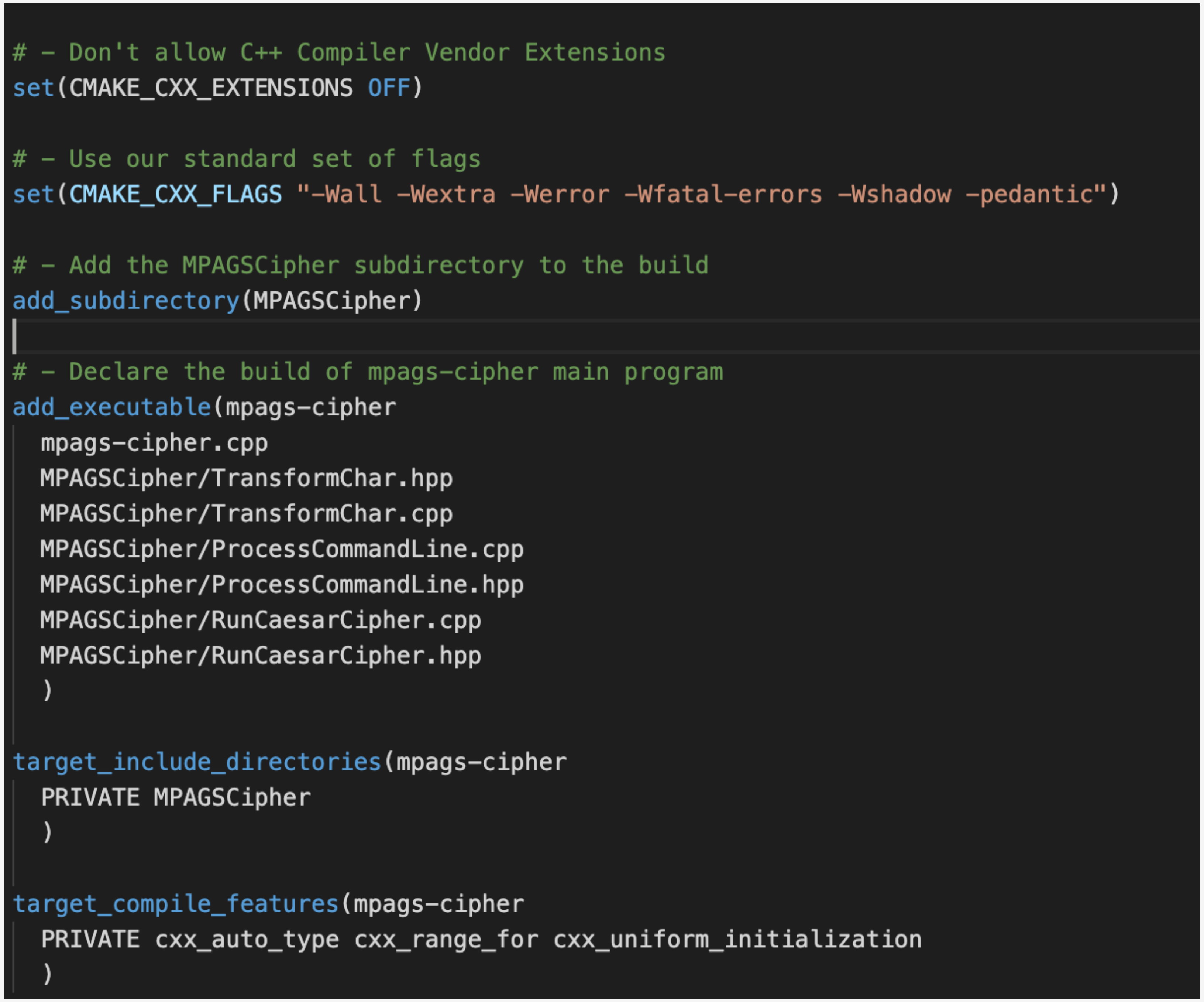
Notes

Again, this choice of structure is arbitrary, but is a common pattern used by projects.

2: Adding MPAGSCipher/ To The Build

We can add a subdirectory to a CMake build by using the add_subdirectory command. It takes the path to the directory holding a further CMakeLists.txt script to be processed as its argument. If the path is relative, it is taken to be relative to the directory holding the CMakeLists.txt in which add_subdirectory was called.

Use add_subdirectory **to add the** MPAGSCipher/ **directory to the build. To confirm it works, try using the CMake** message **command in** MPAGSCipher/CMakeLists.txt



```
# - Don't allow C++ Compiler Vendor Extensions
set(CMAKE_CXX_EXTENSIONS OFF)

# - Use our standard set of flags
set(CMAKE_CXX_FLAGS "-Wall -Wextra -Werror -Wfatal-errors -Wshadow -pedantic")

# - Add the MPAGSCipher subdirectory to the build
add_subdirectory(MPAGSCipher)

# - Declare the build of mpags-cipher main program
add_executable(mpags-cipher
  mpags-cipher.cpp
  MPAGSCipher/TransformChar.hpp
  MPAGSCipher/TransformChar.cpp
  MPAGSCipher/ProcessCommandLine.cpp
  MPAGSCipher/ProcessCommandLine.hpp
  MPAGSCipher/RunCaesarCipher.cpp
  MPAGSCipher/RunCaesarCipher.hpp
  )

target_include_directories(mpags-cipher
  PRIVATE MPAGSCipher
  )

target_compile_features(mpags-cipher
  PRIVATE cxx_auto_type cxx_range_for cxx_uniform_initialization
  )
```

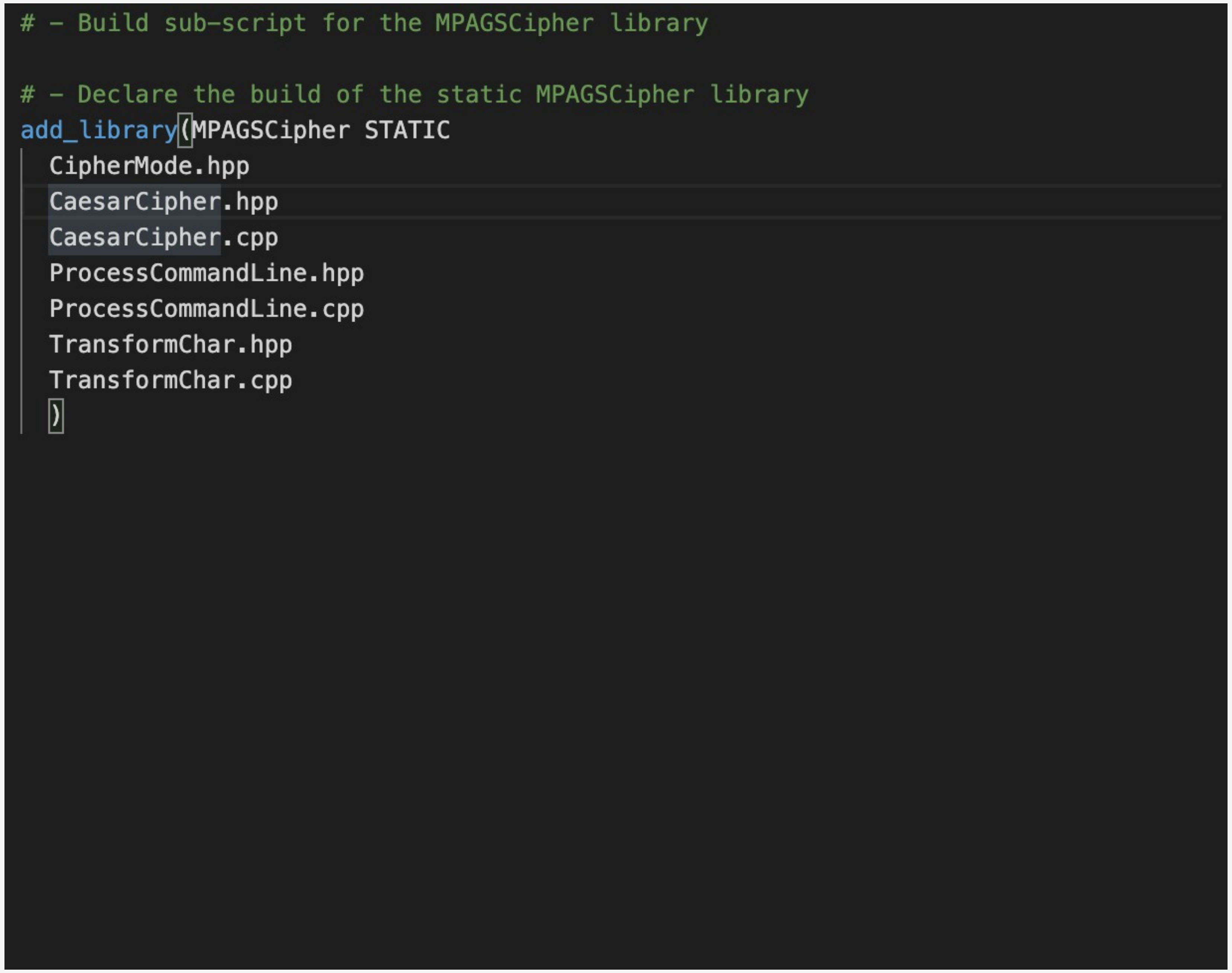
Notes

We only have a single level of subdirectories, but more can be used if required.

3: Building The MPAGSCipher Library

To build a library in CMake, we use its add_library command. This takes the name you want the library to have, the type of library it should be and a space separated list of all the sources that need to be compiled to create the library.

In MPAGSCipher/CMakeLists.txt**, use** add_library **to build a library named** MPAGSCipher**. Use the** STATIC **library type and list the sources that should be compiled to create the library. Take care to specify the correct paths to the sources.**



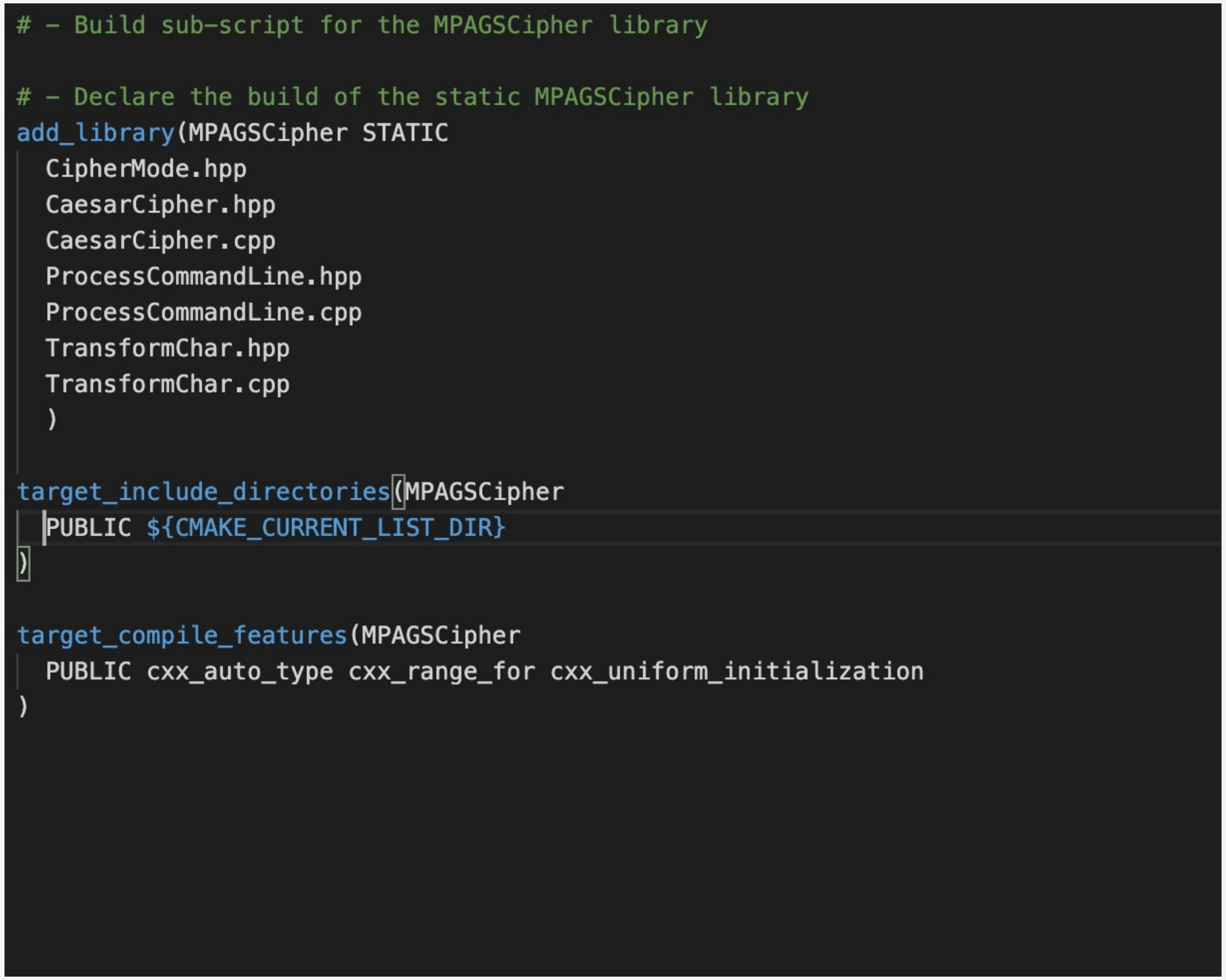
```
# - Build sub-script for the MPAGSCipher library

# - Declare the build of the static MPAGSCipher library
add_library(MPAGSCipher STATIC
  CipherMode.hpp
  CaesarCipher.hpp
  CaesarCipher.cpp
  ProcessCommandLine.hpp
  ProcessCommandLine.cpp
  TransformChar.hpp
  TransformChar.cpp
  )
```

Library Types

There are two main types on library - static and shared. The difference between these is that static libraries have to be built and linked **into** the executable whereas shared libraries are 'referenced' and are shipped as separate files. There are pros and cons to both depending on the situation but here, we will stick with the simpler **static** library.

4: Adding Compile Features and Include Paths

Just as we did for the mpags-cipher **executable, use** target_compile_features **and** target_include_directories **to declare needed C++ features and header search paths for** MPAGSCipher. Both executables and libraries are "targets" in CMake parlance so we can use exactly the same command. **This time, declare the features and paths using the** PUBLIC **scope specifier. We do this because we will have** mpags-cipher **as a client of the library, so it needs to know about these**



```
# - Build sub-script for the MPAGSCipher library

# - Declare the build of the static MPAGSCipher library
add_library(MPAGSCipher STATIC
  CipherMode.hpp
  CaesarCipher.hpp
  CaesarCipher.cpp
  ProcessCommandLine.hpp
  ProcessCommandLine.cpp
  TransformChar.hpp
  TransformChar.cpp
  )

target_include_directories(MPAGSCipher
  PUBLIC ${CMAKE_CURRENT_LIST_DIR}
)

target_compile_features(MPAGSCipher
  PUBLIC cxx_auto_type cxx_range_for cxx_uniform_initialization
)
```

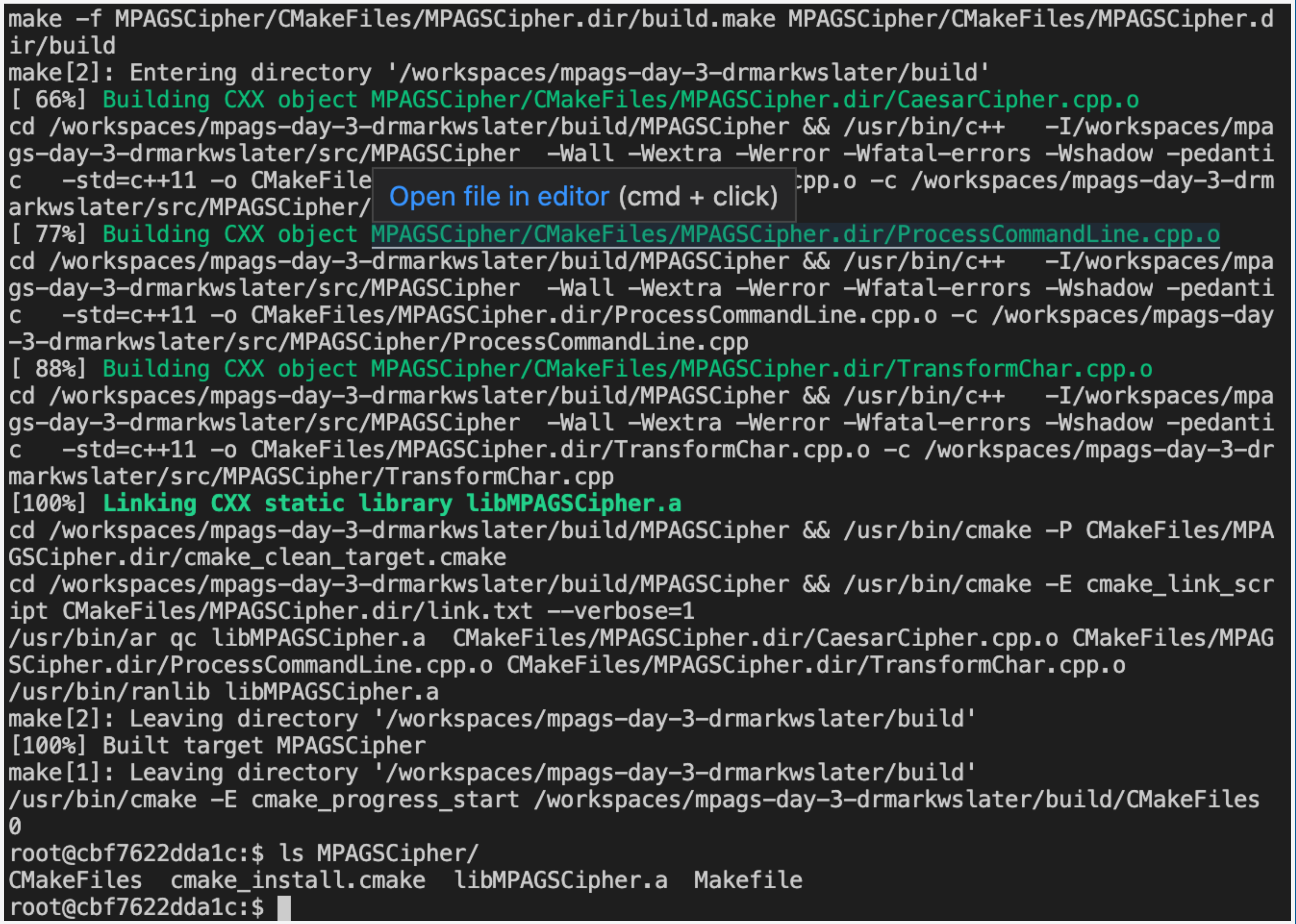
Hints

The path passed to target_include_directories can use the CMake convenience variable CMAKE_CURRENT_LIST_DIR. This has a value equal to the absolute path to the directory holding the CMake script currently being processed.

5: What CMake Has Built

After adding compile features and include directories, try rebuilding and you should see it complete without error (if not, resolve any errors until it does).

The output of our add_library **call is a static library - a file named** libMPAGSCipher.a **which is located under the** MPAGSCipher **subdirectory of the build directory. CMake outputs build products in the same directory structure as used in the source project.**



```
make -f MPAGSCipher/CMakeFiles/MPAGSCipher.dir/build.make MPAGSCipher/CMakeFiles/MPAGSCipher.d
ir/build
make[2]: Entering directory '/workspaces/mpags-day-3-drmarkwslater/build'
[ 66%] Building CXX object MPAGSCipher/CMakeFiles/MPAGSCipher.dir/CaesarCipher.cpp.o
cd /workspaces/mpags-day-3-drmarkwslater/build/MPAGSCipher && /usr/bin/c++   -I/workspaces/mpa
gs-day-3-drmarkwslater/src/MPAGSCipher  -Wall -Wextra -Werror -Wfatal-errors -Wshadow -pedanti
c   -std=c++11 -o CMakeFile                          cpp.o -c /workspaces/mpags-day-3-drm
arkwslater/src/MPAGSCipher/
[ 77%] Building CXX object MPAGSCipher/CMakeFiles/MPAGSCipher.dir/ProcessCommandLine.cpp.o
cd /workspaces/mpags-day-3-drmarkwslater/build/MPAGSCipher && /usr/bin/c++   -I/workspaces/mpa
gs-day-3-drmarkwslater/src/MPAGSCipher  -Wall -Wextra -Werror -Wfatal-errors -Wshadow -pedanti
c   -std=c++11 -o CMakeFiles/MPAGSCipher.dir/ProcessCommandLine.cpp.o -c /workspaces/mpags-day
-3-drmarkwslater/src/MPAGSCipher/ProcessCommandLine.cpp
[ 88%] Building CXX object MPAGSCipher/CMakeFiles/MPAGSCipher.dir/TransformChar.cpp.o
cd /workspaces/mpags-day-3-drmarkwslater/build/MPAGSCipher && /usr/bin/c++   -I/workspaces/mpa
gs-day-3-drmarkwslater/src/MPAGSCipher  -Wall -Wextra -Werror -Wfatal-errors -Wshadow -pedanti
c   -std=c++11 -o CMakeFiles/MPAGSCipher.dir/TransformChar.cpp.o -c /workspaces/mpags-day-3-dr
markwslater/src/MPAGSCipher/TransformChar.cpp
[100%] Linking CXX static library libMPAGSCipher.a
cd /workspaces/mpags-day-3-drmarkwslater/build/MPAGSCipher && /usr/bin/cmake -P CMakeFiles/MPA
GSCipher.dir/cmake_clean_target.cmake
cd /workspaces/mpags-day-3-drmarkwslater/build/MPAGSCipher && /usr/bin/cmake -E cmake_link_scr
ipt CMakeFiles/MPAGSCipher.dir/link.txt --verbose=1
/usr/bin/ar qc libMPAGSCipher.a  CMakeFiles/MPAGSCipher.dir/CaesarCipher.cpp.o CMakeFiles/MPAG
SCipher.dir/ProcessCommandLine.cpp.o CMakeFiles/MPAGSCipher.dir/TransformChar.cpp.o
/usr/bin/ranlib libMPAGSCipher.a
make[2]: Leaving directory '/workspaces/mpags-day-3-drmarkwslater/build'
[100%] Built target MPAGSCipher
make[1]: Leaving directory '/workspaces/mpags-day-3-drmarkwslater/build'
/usr/bin/cmake -E cmake_progress_start /workspaces/mpags-day-3-drmarkwslater/build/CMakeFiles
0
root@cbf7622dda1c:$ ls MPAGSCipher/
CMakeFiles  cmake_install.cmake  libMPAGSCipher.a  Makefile
root@cbf7622dda1c:$
```

Library Names

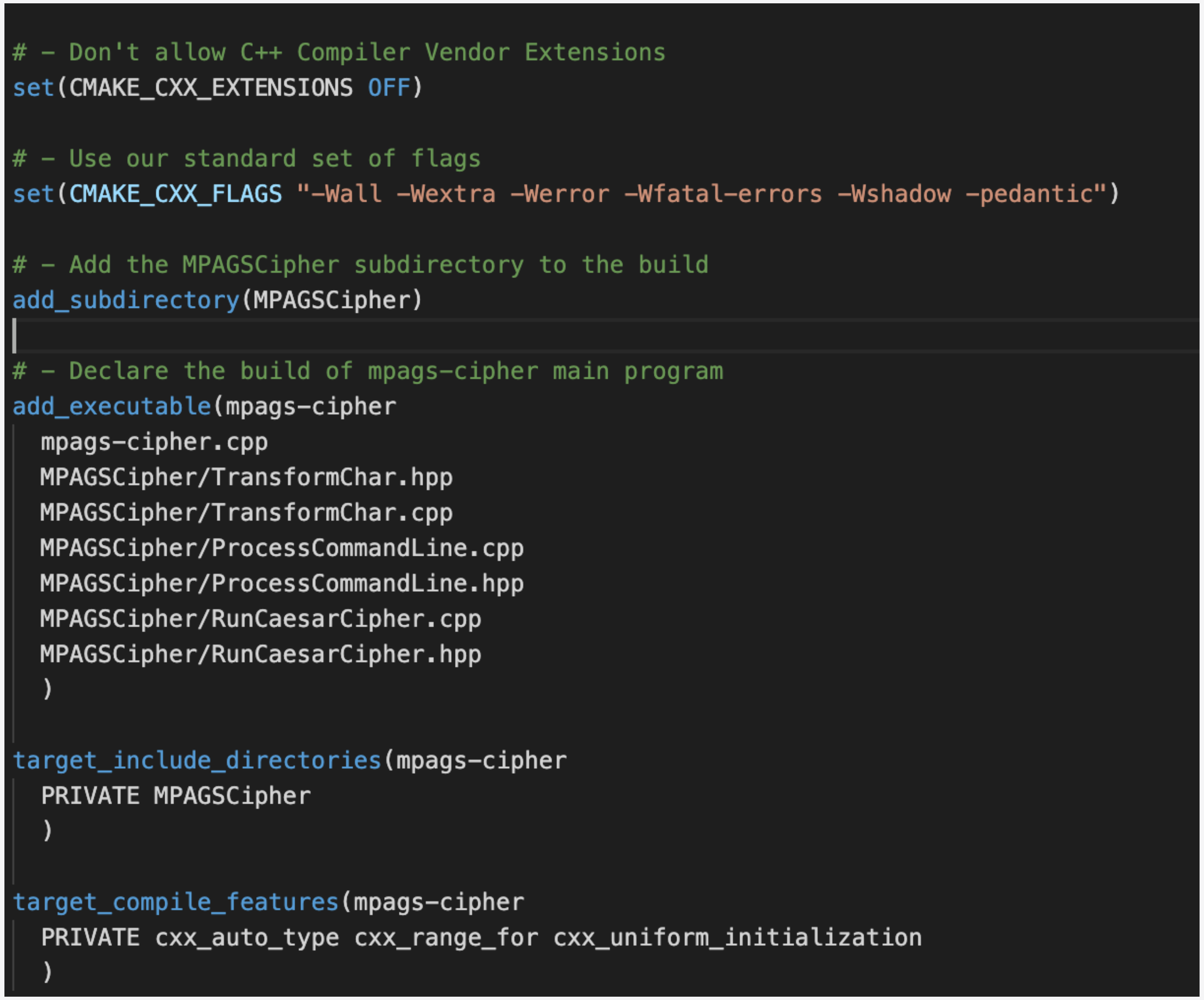
The UNIX convention is to name libraries libNAME.EXT. NAME is as you might guess, EXT is 'a' for static libraries, but 'so' for shared libraries, except on OS X where 'dylib' is used.

On Windows, NAME.EXT is used, with EXT being 'dll' for shared libraries, and '.lib' for static/import libraries.

6: Using The MPAGSCipher Library

As things stand, we're still compiling all sources under MPAGSCipher twice - once for the mpags-cipher executable and once for the MPAGSCipher library. With the latter now built, we can remove its sources from the add_executable call for mpags-cipher and instead link mpags-cipher to the MPAGSCipher library.

In CMake, we link a target to libraries using the target_link_libraries **command, and we'll see how to use this next**



```
# - Don't allow C++ Compiler Vendor Extensions
set(CMAKE_CXX_EXTENSIONS OFF)

# - Use our standard set of flags
set(CMAKE_CXX_FLAGS "-Wall -Wextra -Werror -Wfatal-errors -Wshadow -pedantic")

# - Add the MPAGSCipher subdirectory to the build
add_subdirectory(MPAGSCipher)

# - Declare the build of mpags-cipher main program
add_executable(mpags-cipher
  mpags-cipher.cpp
  MPAGSCipher/TransformChar.hpp
  MPAGSCipher/TransformChar.cpp
  MPAGSCipher/ProcessCommandLine.cpp
  MPAGSCipher/ProcessCommandLine.hpp
  MPAGSCipher/RunCaesarCipher.cpp
  MPAGSCipher/RunCaesarCipher.hpp
  )

target_include_directories(mpags-cipher
  PRIVATE MPAGSCipher
  )

target_compile_features(mpags-cipher
  PRIVATE cxx_auto_type cxx_range_for cxx_uniform_initialization
  )
```

Notes

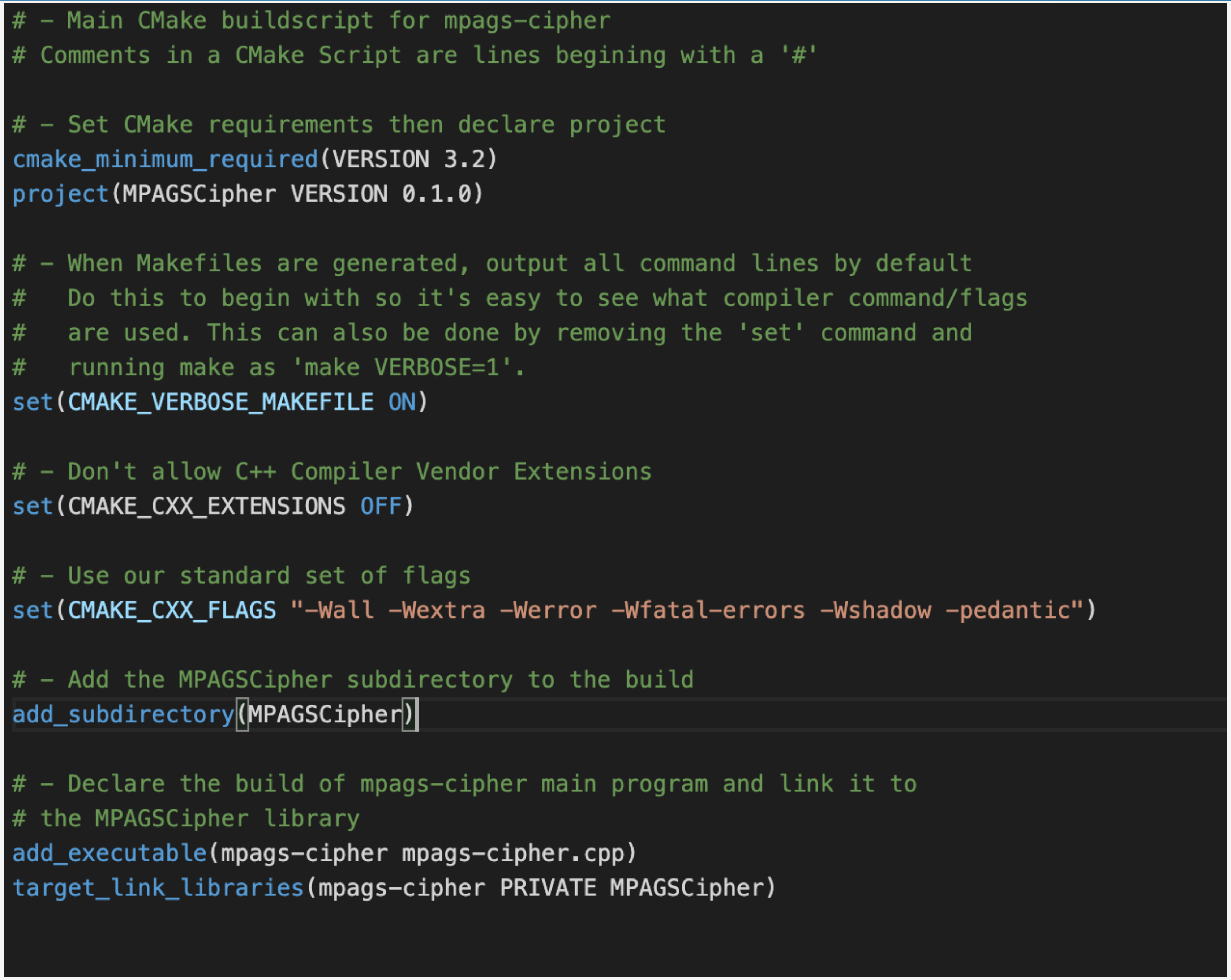
There are occasional use cases where you may need to compile the same file more than once.

CMake will handle this as we have seen, but in general the use cases are quite advanced.

7: Using target_link_libraries

The target_link_libraries command takes the name of the target requiring linking, a link scope specifier and a list of targets to be linked to it.

In your top level CMakeLists.txt, **build** mpags-cipher **from** mpags-cipher.cpp **only. Replace the calls to** target_compile_features **and** target_include_directories **with a call to** target_link_libraries **that links the** mpags-cipher **executable to the** MPAGSCipher **library using the** PRIVATE **scope specifier.**



```
# - Main CMake buildscript for mpags-cipher
# Comments in a CMake Script are lines begining with a '#'

# - Set CMake requirements then declare project
cmake_minimum_required(VERSION 3.2)
project(MPAGSCipher VERSION 0.1.0)

# - When Makefiles are generated, output all command lines by default
#   Do this to begin with so it's easy to see what compiler command/flags
#   are used. This can also be done by removing the 'set' command and
#   running make as 'make VERBOSE=1'.
set(CMAKE_VERBOSE_MAKEFILE ON)

# - Don't allow C++ Compiler Vendor Extensions
set(CMAKE_CXX_EXTENSIONS OFF)

# - Use our standard set of flags
set(CMAKE_CXX_FLAGS "-Wall -Wextra -Werror -Wfatal-errors -Wshadow -pedantic")

# - Add the MPAGSCipher subdirectory to the build
add_subdirectory(MPAGSCipher)

# - Declare the build of mpags-cipher main program and link it to
# the MPAGSCipher library
add_executable(mpags-cipher mpags-cipher.cpp)
target_link_libraries(mpags-cipher PRIVATE MPAGSCipher)
```

Notes

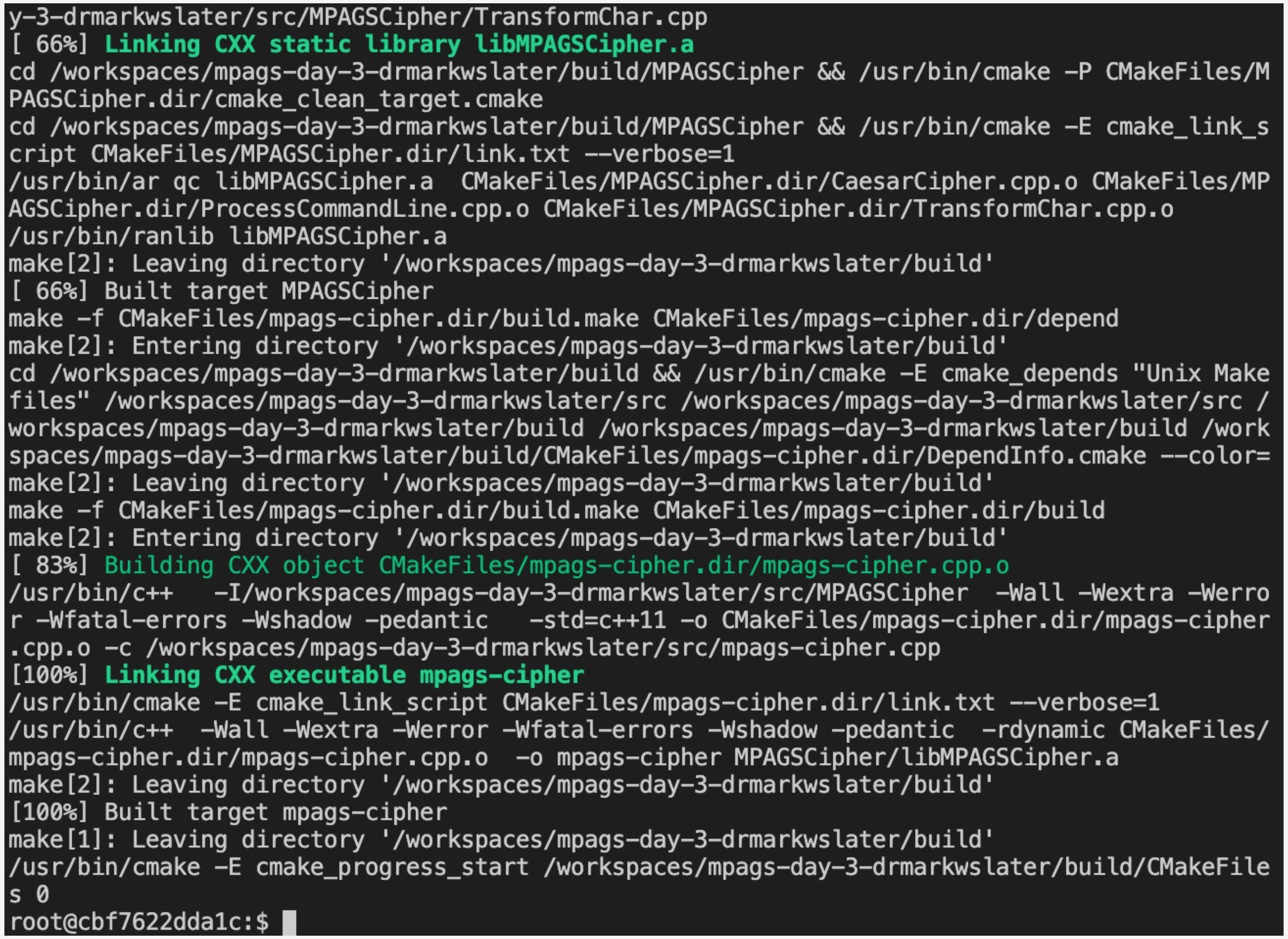
Why remove target_compile_features and target_include_directories? See the next slide!

We've used the PRIVATE scope specifier here as mpags-cipher is the end point of the build process. We don't link anything to it. so nothing needs to know that it uses MPAGSCipher.

8: Rebuilding mpags-cipher

After you've edited your CMake script for mpags-cipher, rebuild and note what happens.

You should find that compilation of mpags-cipher.cpp **has all the correct flags, including** -std=c++11 **and the include path. By specifying the compile features and include paths of** MPAGSCipher **as** PUBLIC **scope, we can simply link to it and CMake will take care of setting required compile features and include paths for us! You'll also see that the** libMPAGSCipher.a **file is added to the linking step as expected.**



```
y-3-drmarkwslater/src/MPAGSCipher/TransformChar.cpp
[ 66%] Linking CXX static library libMPAGSCipher.a
cd /workspaces/mpags-day-3-drmarkwslater/build/MPAGSCipher && /usr/bin/cmake -P CMakeFiles/M
PAGSCipher.dir/cmake_clean_target.cmake
cd /workspaces/mpags-day-3-drmarkwslater/build/MPAGSCipher && /usr/bin/cmake -E cmake_link_s
cript CMakeFiles/MPAGSCipher.dir/link.txt --verbose=1
/usr/bin/ar qc libMPAGSCipher.a  CMakeFiles/MPAGSCipher.dir/CaesarCipher.cpp.o CMakeFiles/MP
AGSCipher.dir/ProcessCommandLine.cpp.o CMakeFiles/MPAGSCipher.dir/TransformChar.cpp.o
/usr/bin/ranlib libMPAGSCipher.a
make[2]: Leaving directory '/workspaces/mpags-day-3-drmarkwslater/build'
[ 66%] Built target MPAGSCipher
make -f CMakeFiles/mpags-cipher.dir/build.make CMakeFiles/mpags-cipher.dir/depend
make[2]: Entering directory '/workspaces/mpags-day-3-drmarkwslater/build'
cd /workspaces/mpags-day-3-drmarkwslater/build && /usr/bin/cmake -E cmake_depends "Unix Make
files" /workspaces/mpags-day-3-drmarkwslater/src /workspaces/mpags-day-3-drmarkwslater/src /
workspaces/mpags-day-3-drmarkwslater/build /workspaces/mpags-day-3-drmarkwslater/build /work
spaces/mpags-day-3-drmarkwslater/build/CMakeFiles/mpags-cipher.dir/DependInfo.cmake --color=
make[2]: Leaving directory '/workspaces/mpags-day-3-drmarkwslater/build'
make -f CMakeFiles/mpags-cipher.dir/build.make CMakeFiles/mpags-cipher.dir/build
make[2]: Entering directory '/workspaces/mpags-day-3-drmarkwslater/build'
[ 83%] Building CXX object CMakeFiles/mpags-cipher.dir/mpags-cipher.cpp.o
/usr/bin/c++   -I/workspaces/mpags-day-3-drmarkwslater/src/MPAGSCipher  -Wall -Wextra -Werro
r -Wfatal-errors -Wshadow -pedantic   -std=c++11 -o CMakeFiles/mpags-cipher.dir/mpags-cipher
.cpp.o -c /workspaces/mpags-day-3-drmarkwslater/src/mpags-cipher.cpp
[100%] Linking CXX executable mpags-cipher
/usr/bin/cmake -E cmake_link_script CMakeFiles/mpags-cipher.dir/link.txt --verbose=1
/usr/bin/c++   -Wall -Wextra -Werror -Wfatal-errors -Wshadow -pedantic   -rdynamic CMakeFiles/
mpags-cipher.dir/mpags-cipher.cpp.o  -o mpags-cipher MPAGSCipher/libMPAGSCipher.a
make[2]: Leaving directory '/workspaces/mpags-day-3-drmarkwslater/build'
[100%] Built target mpags-cipher
make[1]: Leaving directory '/workspaces/mpags-day-3-drmarkwslater/build'
/usr/bin/cmake -E cmake_progress_start /workspaces/mpags-day-3-drmarkwslater/build/CMakeFile
s 0
root@cbf7622dda1c:$
```

Notes

We could add extra compile features and include paths to mpags-cipher if we need them. CMake will simply merge them with those of any target linked.

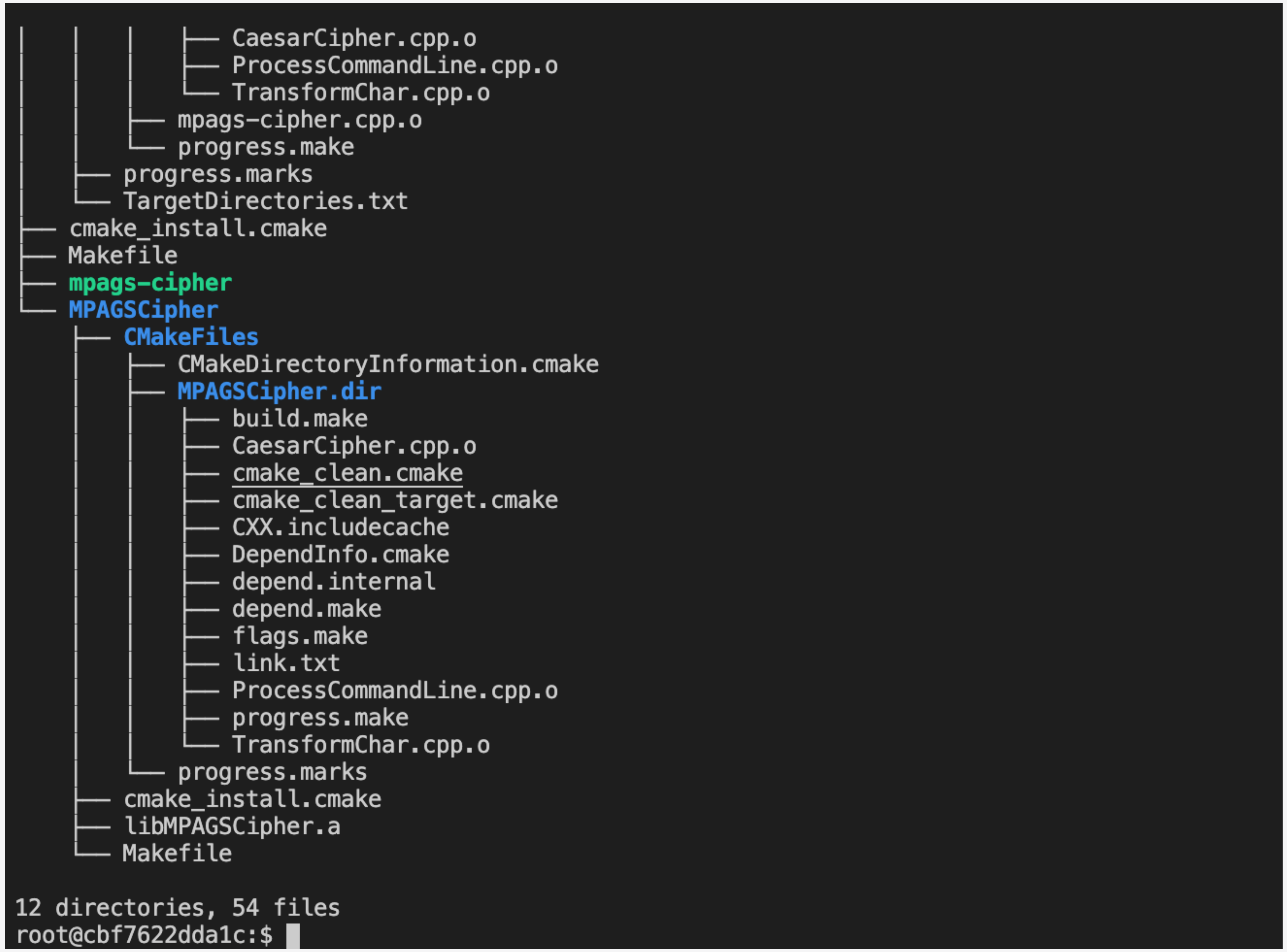
The library is treated by the linker just like any other object file, so it is simply added as an extra input in the link step.

9: Library Build Summary

In this first part, we've partitioned the build of mpags-cipher into an executable linked to a library, the latter holding the major part of the implementation such as functions.

We've done this so we can use that implementation in several places, in this case we can now link that library to other executables that'll test units of that implementation.

Use of a build system like CMake has made that partition easy.



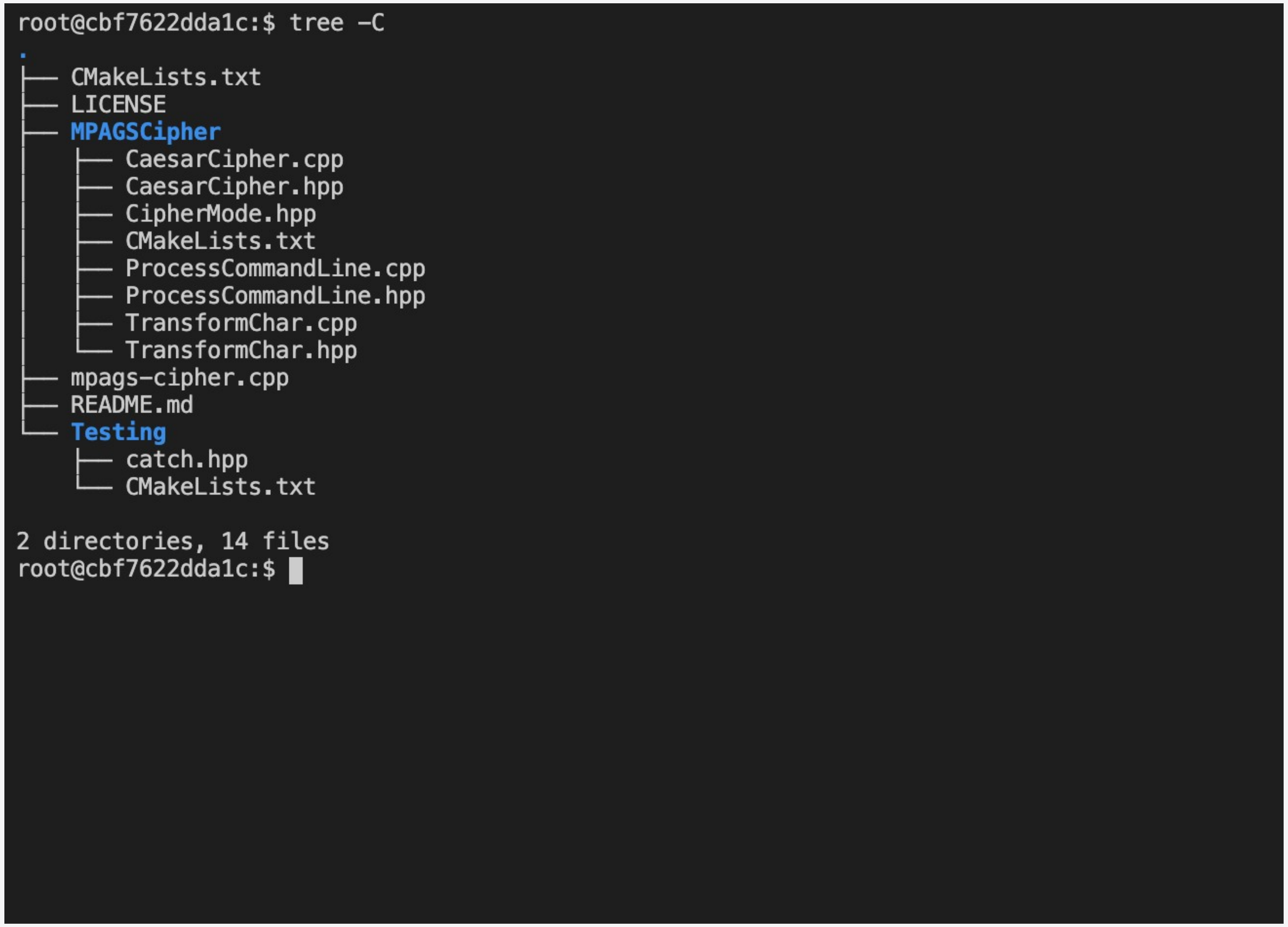
```
│   │   │   ├── CaesarCipher.cpp.o
│   │   │   ├── ProcessCommandLine.cpp.o
│   │   │   └── TransformChar.cpp.o
│   │   ├── mpags-cipher.cpp.o
│   │   └── progress.make
│   ├── progress.marks
│   └── TargetDirectories.txt
├── cmake_install.cmake
├── Makefile
├── mpags-cipher
└── MPAGSCipher
    ├── CMakeFiles
    │   ├── CMakeDirectoryInformation.cmake
    │   ├── MPAGSCipher.dir
    │   │   ├── build.make
    │   │   ├── CaesarCipher.cpp.o
    │   │   ├── cmake_clean.cmake
    │   │   ├── cmake_clean_target.cmake
    │   │   ├── CXX.includecache
    │   │   ├── DependInfo.cmake
    │   │   ├── depend.internal
    │   │   ├── depend.make
    │   │   ├── flags.make
    │   │   ├── link.txt
    │   │   ├── ProcessCommandLine.cpp.o
    │   │   ├── progress.make
    │   │   └── TransformChar.cpp.o
    │   └── progress.marks
    ├── cmake_install.cmake
    ├── libMPAGSCipher.a
    └── Makefile

12 directories, 54 files
root@cbf7622dda1c:$
```

Notes

If you don't have a working build at this point, check with us!

10: Project Structure for Testing

There are no hard and fast rules for where to store unit test code in a project. **For clarity, it's usually best to store the files in a subdirectory of the main project, and we'll do this in** mpags-cipher **with the** Testing **subdirectory. As mentioned, this directory is already present in the root of the** mpags-cipher **source tree. Within this, create a blank** CMakeLists.txt **file.**



```
root@cbf7622dda1c:$ tree -C
.
├── CMakeLists.txt
├── LICENSE
├── MPAGSCipher
│   ├── CaesarCipher.cpp
│   ├── CaesarCipher.hpp
│   ├── CipherMode.hpp
│   ├── CMakeLists.txt
│   ├── ProcessCommandLine.cpp
│   ├── ProcessCommandLine.hpp
│   ├── TransformChar.cpp
│   └── TransformChar.hpp
├── mpags-cipher.cpp
├── README.md
└── Testing
    ├── catch.hpp
    └── CMakeLists.txt

2 directories, 14 files
root@cbf7622dda1c:$
```

Notes

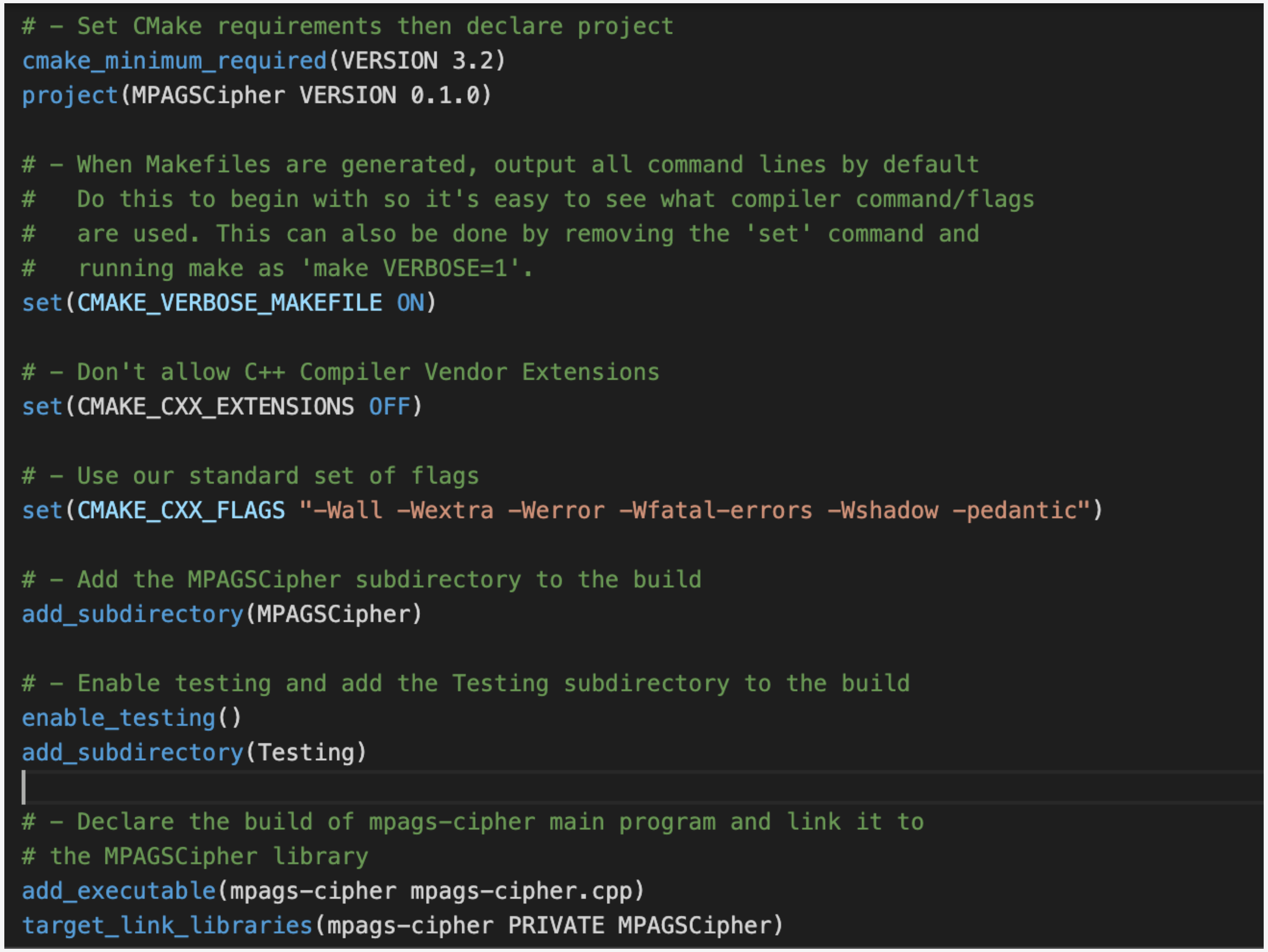
This structure is usually followed in other languages such as Python.

There's nothing to stop test code being alongside the code it's testing. It can cause a little confusion over what is implementation and what is test though, plus naming clashes might occur.

11: Enabling Testing in CMake

CMake provides a basic structure for adding and running test programs as part of the generated build system using its ctest program (you can find this alongside the cmake program).

To use this functionality, add a call to the enable_testing() **command in your top level CMake script. Following this, we also need to make CMake aware of the** Testing **subdirectory, so also recurse the build into this using** add_subdirectory **as we did for the MPAGSCipher directory.**



```
# - Set CMake requirements then declare project
cmake_minimum_required(VERSION 3.2)
project(MPAGSCipher VERSION 0.1.0)

# - When Makefiles are generated, output all command lines by default
#   Do this to begin with so it's easy to see what compiler command/flags
#   are used. This can also be done by removing the 'set' command and
#   running make as 'make VERBOSE=1'.
set(CMAKE_VERBOSE_MAKEFILE ON)

# - Don't allow C++ Compiler Vendor Extensions
set(CMAKE_CXX_EXTENSIONS OFF)

# - Use our standard set of flags
set(CMAKE_CXX_FLAGS "-Wall -Wextra -Werror -Wfatal-errors -Wshadow -pedantic")

# - Add the MPAGSCipher subdirectory to the build
add_subdirectory(MPAGSCipher)

# - Enable testing and add the Testing subdirectory to the build
enable_testing()
add_subdirectory(Testing)

# - Declare the build of mpags-cipher main program and link it to
# the MPAGSCipher library
add_executable(mpags-cipher mpags-cipher.cpp)
target_link_libraries(mpags-cipher PRIVATE MPAGSCipher)
```

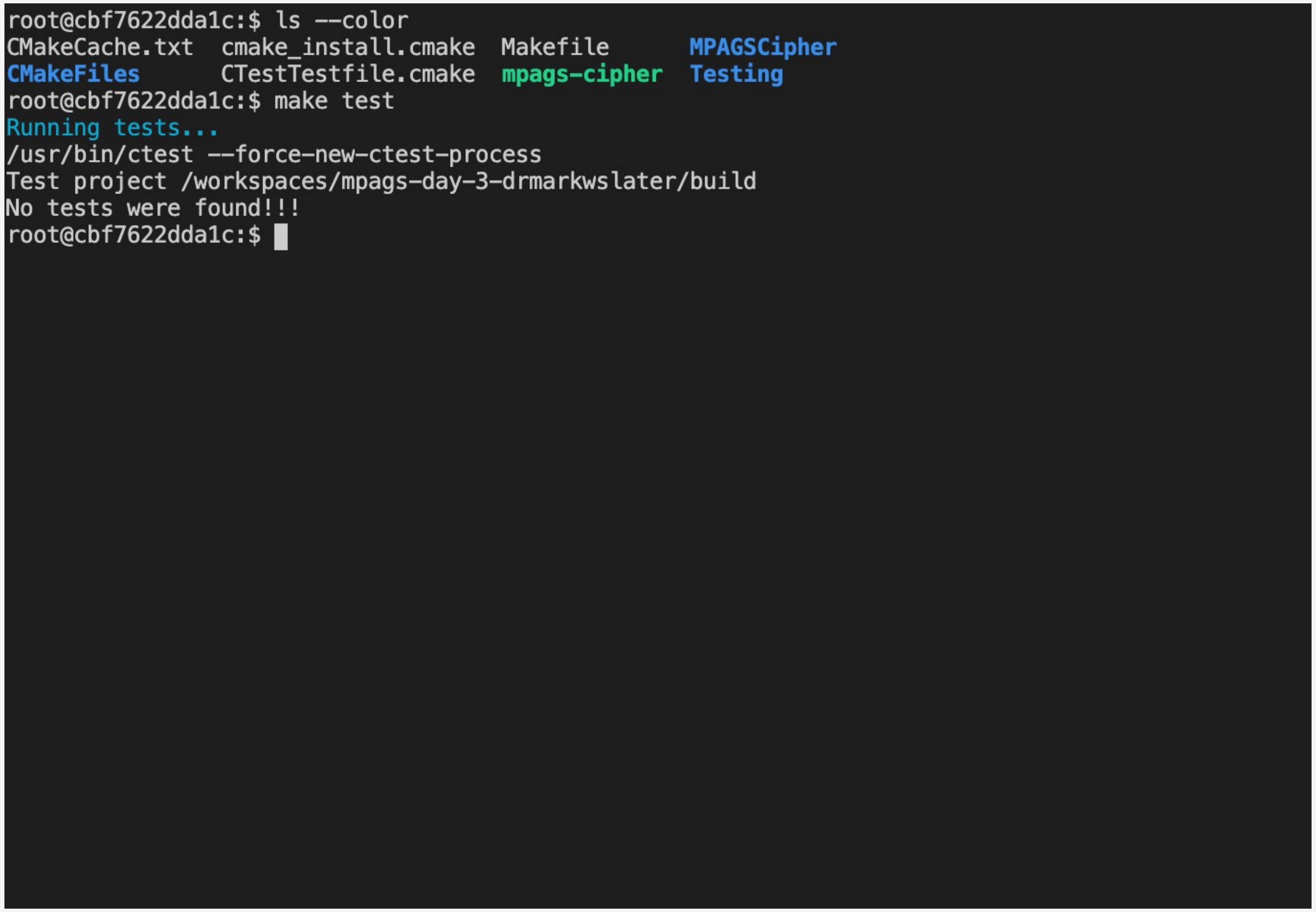
Notes

The call to enable_testing() must be made in the top level CMake script of the project, no matter where the tests actually are.

12: Running Tests

Even though we don't have any tests implemented yet, we can check that everything's set up correctly and see how they'll be run.

Move back to your build directory and re-run cmake and/or make as needed. You should see that a new file CTestTestfile.cmake has been created, and a new Make target test is available. Try "building" this with make test and not much happens as we don't have any tests yet, but this is how we'll run the tests when we have them



```
root@cbf7622dda1c:$ ls --color
CMakeCache.txt  cmake_install.cmake  Makefile      MPAGSCipher
CMakeFiles      CTestTestfile.cmake  mpags-cipher  Testing
root@cbf7622dda1c:$ make test
Running tests...
/usr/bin/ctest --force-new-ctest-process
Test project /workspaces/mpags-day-3-drmarkwslater/build
No tests were found!!!
root@cbf7622dda1c:$
```

Notes

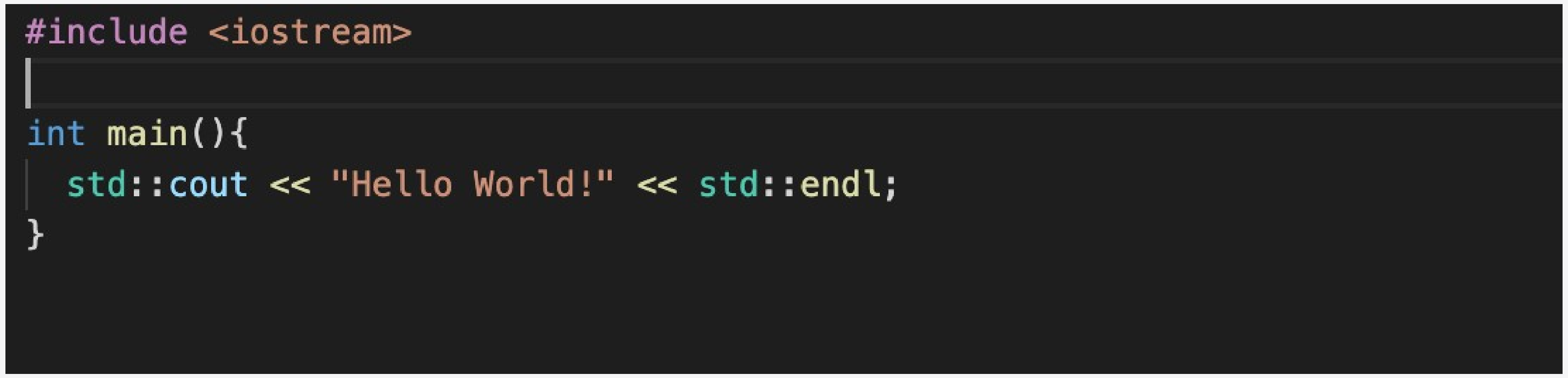
A similar test target will also be created in IDEs like Xcode.

This target simply runs the ctest program, and you can see this by running ctest directly in your build directory. To get verbose output, run ctest -V

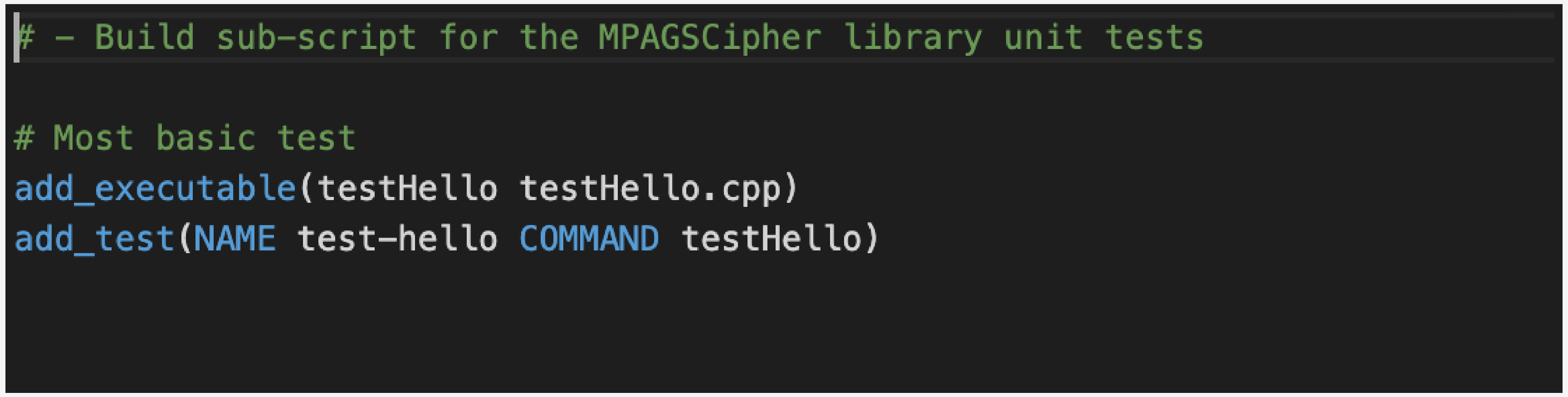
13: Adding a New Test Program

To define a test in CMake, we first use add_executable to build the test program, then add_test to declare a new test using this program as the command to run.

To start, write testHello.cpp **in** Testing **with the basic "hello world" in C++. Use** add_executable **in** Testing/CMakeLists.txt **to build a** testHello **program from it, then use** add_test **to make it the command of a test named** test-hello



```cpp
#include <iostream>

int main(){
  std::cout << "Hello World!" << std::endl;
}
```



```cmake
# - Build sub-script for the MPAGSCipher library unit tests

# Most basic test
add_executable(testHello testHello.cpp)
add_test(NAME test-hello COMMAND testHello)
```

Notes

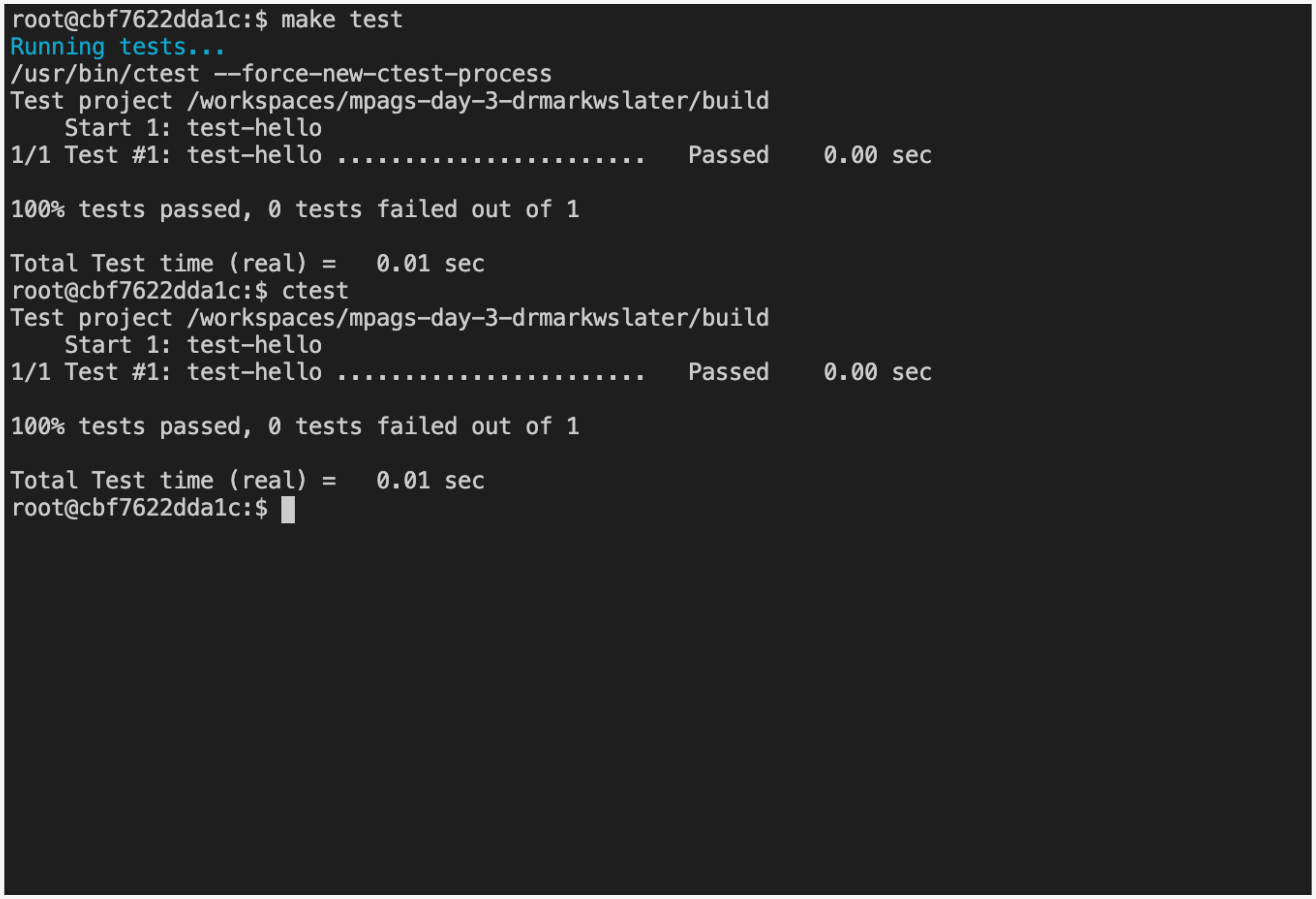
This extremely simple use of add_test is all we'll need in this course.

More advanced usage is enabled via test properties. These include things like maximum runtime and dependencies between tests (e.g. one generates a file used by another).

14: Building and Running the Test Program

Test executables are built as part of the main build task, so simply rerun this (make in this case) to rebuild - of course the executable should compile!

The tests are run either by "building" the test **target, i.e.** make test, **or by running the** ctest **command directly. Try both and note the differences. Try running** ctest -VV **to get more detailed reporting.**



```
root@cbf7622dda1c:$ make test
Running tests...
/usr/bin/ctest --force-new-ctest-process
Test project /workspaces/mpags-day-3-drmarkwslater/build
    Start 1: test-hello
1/1 Test #1: test-hello .......................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec
root@cbf7622dda1c:$ ctest
Test project /workspaces/mpags-day-3-drmarkwslater/build
    Start 1: test-hello
1/1 Test #1: test-hello .......................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec
root@cbf7622dda1c:$
```

Notes

Note that make test will not rebuild the test executables if they change!

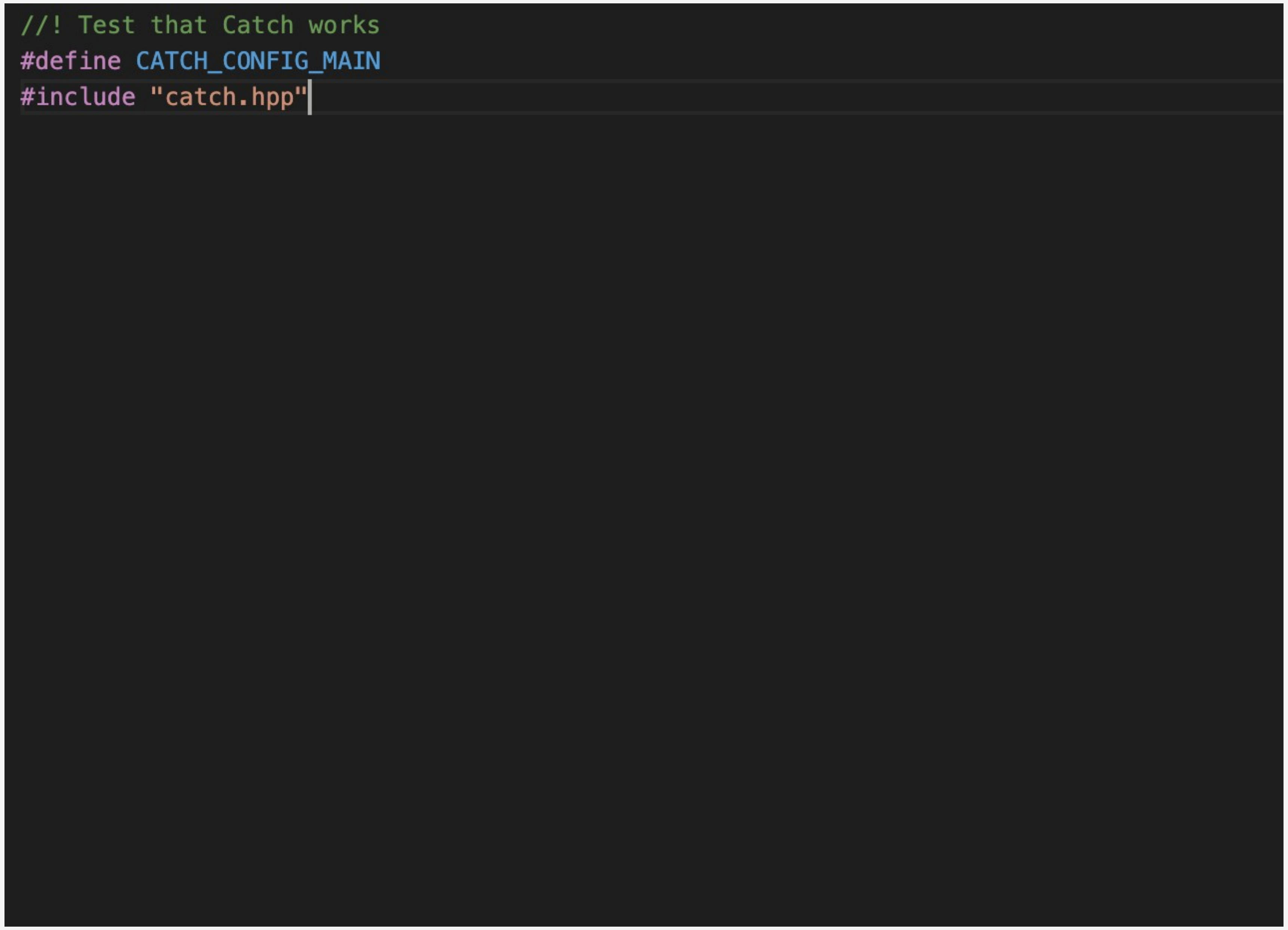
Generally, make test is best for quick checks as you develop. Use of ctest is best when you need more detailed output or to run individual tests to debug.

15: A First Catch-based Test Program

Open a file named testCatch.cpp **in** Testing/ **and add two lines**

#define CATCH_CONFIG_MAIN is a simple preprocessor define to tell Catch to provide a main() function for us. This simplifies writing tests and will also provide several command line options for the resulting executable (just like those we've been writing for mpags-cipher)

#include "catch.hpp" of course just includes the single Catch header



```
//! Test that Catch works
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

Notes

This provision of a main() function by the testing framework is quite common.

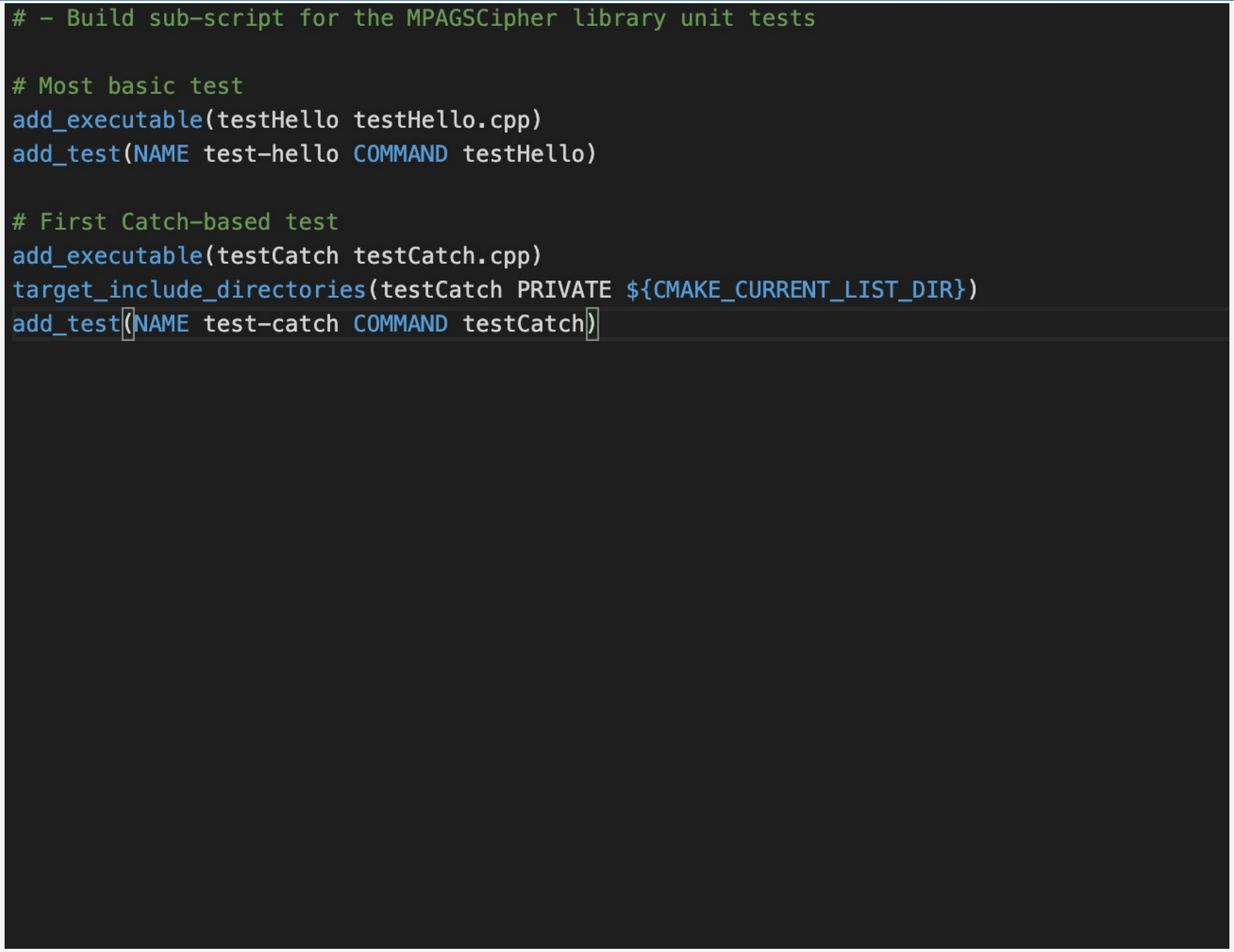
It helps to focus on the task of writing tests, and allows the executable to be provisioned with extra functionality, like command line arguments.

16: Building the Catch-based Test Program

Compile the testCatch.cpp **file into a program named** testCatch**, using** add_executable **to build it, and** target_include_directories **to ensure the** Testing/ **subdirectory is used to find the** Catch.hpp **header.**

Use add_test **to create a test named** test-catch **that runs the** testCatch **program.**

Rerun make **in the build directory, and check that it compiles correctly.**



```
# - Build sub-script for the MPAGSCipher library unit tests

# Most basic test
add_executable(testHello testHello.cpp)
add_test(NAME test-hello COMMAND testHello)

# First Catch-based test
add_executable(testCatch testCatch.cpp)
target_include_directories(testCatch PRIVATE ${CMAKE_CURRENT_LIST_DIR})
add_test(NAME test-catch COMMAND testCatch)
```

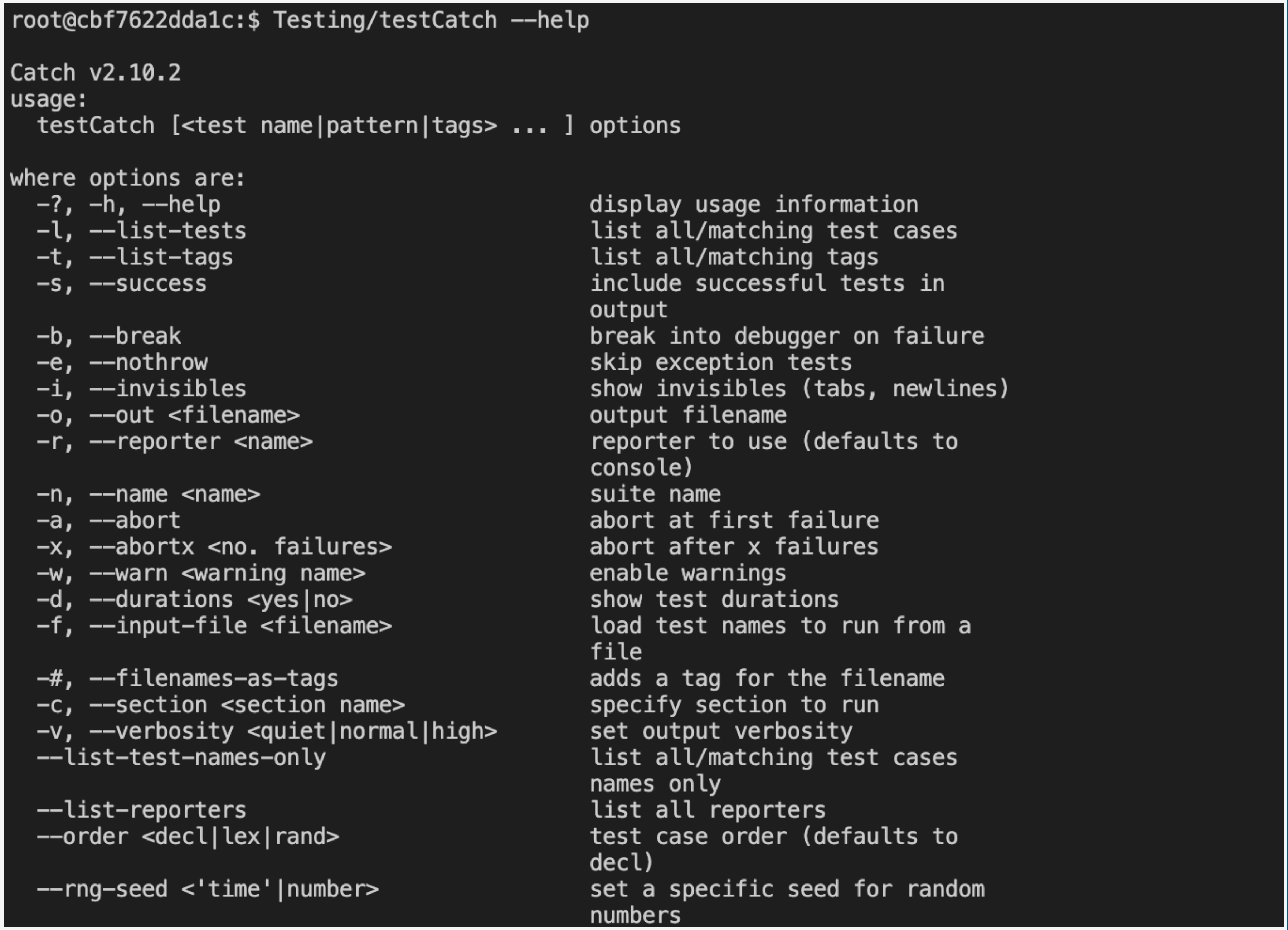
Notes

Don't worry if you find compilation of your Catch program taking a while. The header is large and complex, so this is the small price we pay for ease of use!

17: Running the Catch-based Test Program

As we did for the "hello world" test, once you have testCatch **building correctly, try running it using** make test **and** ctest -VV**.**

Just like other programs, the actual executable is output to the build directory under Testing/testCatch. You can also run this directly, so try this, passing it the --help command line flag. This, and the other listed arguments, are supplied because we got Catch to create main().



```
root@cbf7622dda1c:$ Testing/testCatch --help

Catch v2.10.2
usage:
  testCatch [<test name|pattern|tags> ... ] options

where options are:
  -?, -h, --help                        display usage information
  -l, --list-tests                      list all/matching test cases
  -t, --list-tags                       list all/matching tags
  -s, --success                         include successful tests in
                                        output
  -b, --break                           break into debugger on failure
  -e, --nothrow                         skip exception tests
  -i, --invisibles                      show invisibles (tabs, newlines)
  -o, --out <filename>                  output filename
  -r, --reporter <name>                 reporter to use (defaults to
                                        console)
  -n, --name <name>                     suite name
  -a, --abort                           abort at first failure
  -x, --abortx <no. failures>           abort after x failures
  -w, --warn <warning name>             enable warnings
  -d, --durations <yes|no>              show test durations
  -f, --input-file <filename>           load test names to run from a
                                        file
  -#, --filenames-as-tags               adds a tag for the filename
  -c, --section <section name>          specify section to run
  -v, --verbosity <quiet|normal|high>   set output verbosity
  --list-test-names-only                list all/matching test cases
                                        names only
  --list-reporters                      list all reporters
  --order <decl|lex|rand>               test case order (defaults to
                                        decl)
  --rng-seed <'time'|number>            set a specific seed for random
                                        numbers
```

Notes

If you want to pass arguments to add_test, these can be listed after the command itself, e.g.

```
add_test(test-catch
  COMMAND
    testCatch -s)
```

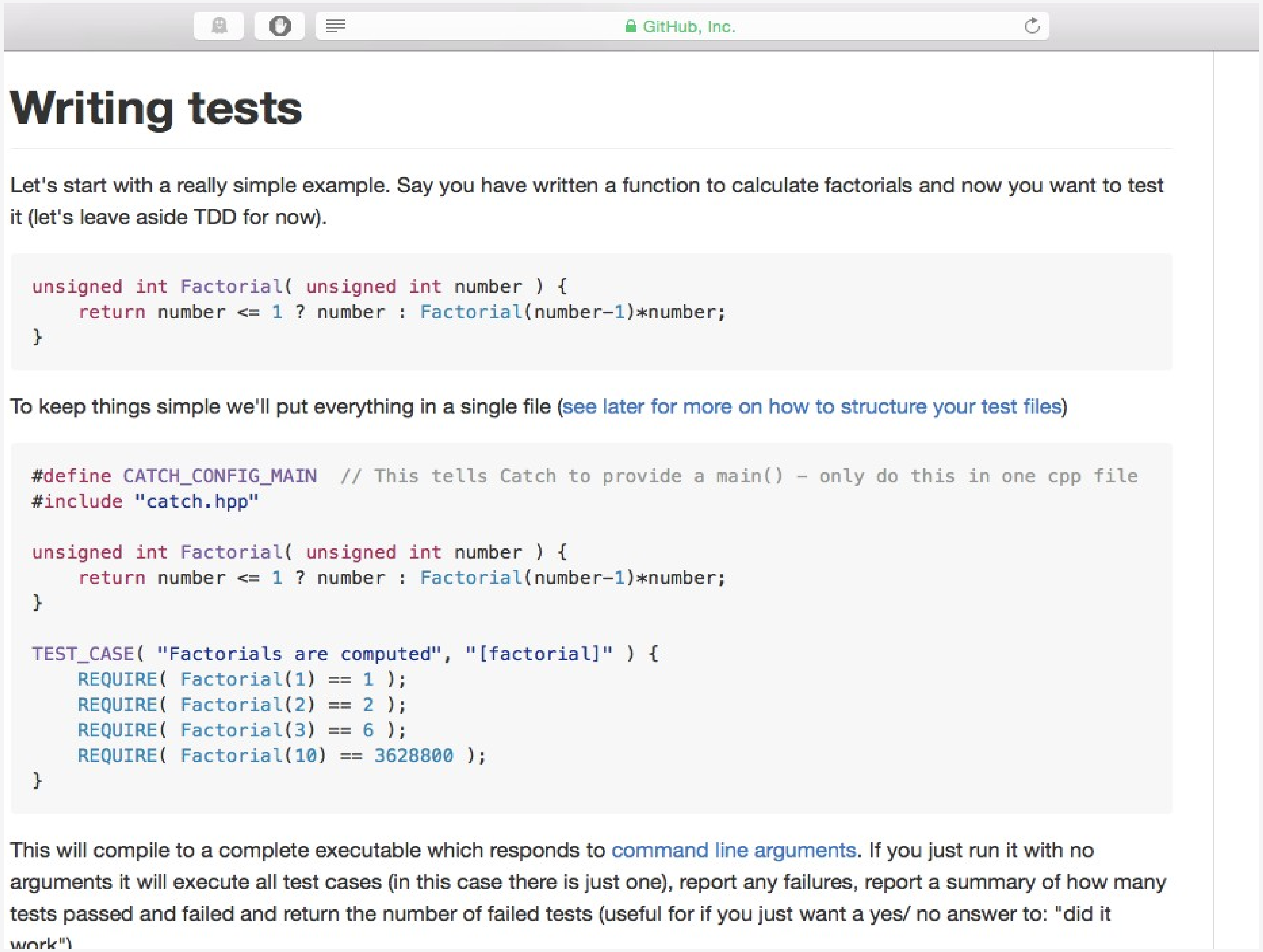
to run testCatch with the argument to output passing and failing tests

18: Test Cases and Assertions

As the “No tests ran” report of testCatch indicates, we haven’t implemented any tests yet.

We’ll add tests using Catch’s TEST_CASE and REQUIRE *macros* - preprocessor “templates” that are expanded at compile time. In this case, we don’t need to worry about this too much and can write and treat them as functions returning void (i.e. nothing)

TEST_CASE **organises tests, whilst** REQUIRE **does the actual test - we supply it with a boolean expression that should evaluate to true if the test passes**


GitHub, Inc.

Writing tests

Let's start with a really simple example. Say you have written a function to calculate factorials and now you want to test it (let's leave aside TDD for now).

```
unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}
```

To keep things simple we'll put everything in a single file (see later for more on how to structure your test files)

```
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main() - only do this in one cpp file
#include "catch.hpp"

unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

This will compile to a complete executable which responds to command line arguments. If you just run it with no arguments it will execute all test cases (in this case there is just one), report any failures, report a summary of how many tests passed and failed and return the number of failed tests (useful for if you just want a yes/ no answer to: "did it work")

Macros vs Functions

Test and other frameworks use macros when the user needs to supply a long but well-defined block of code in which only one or two names (strings, digits, typenames) may need to be changed.

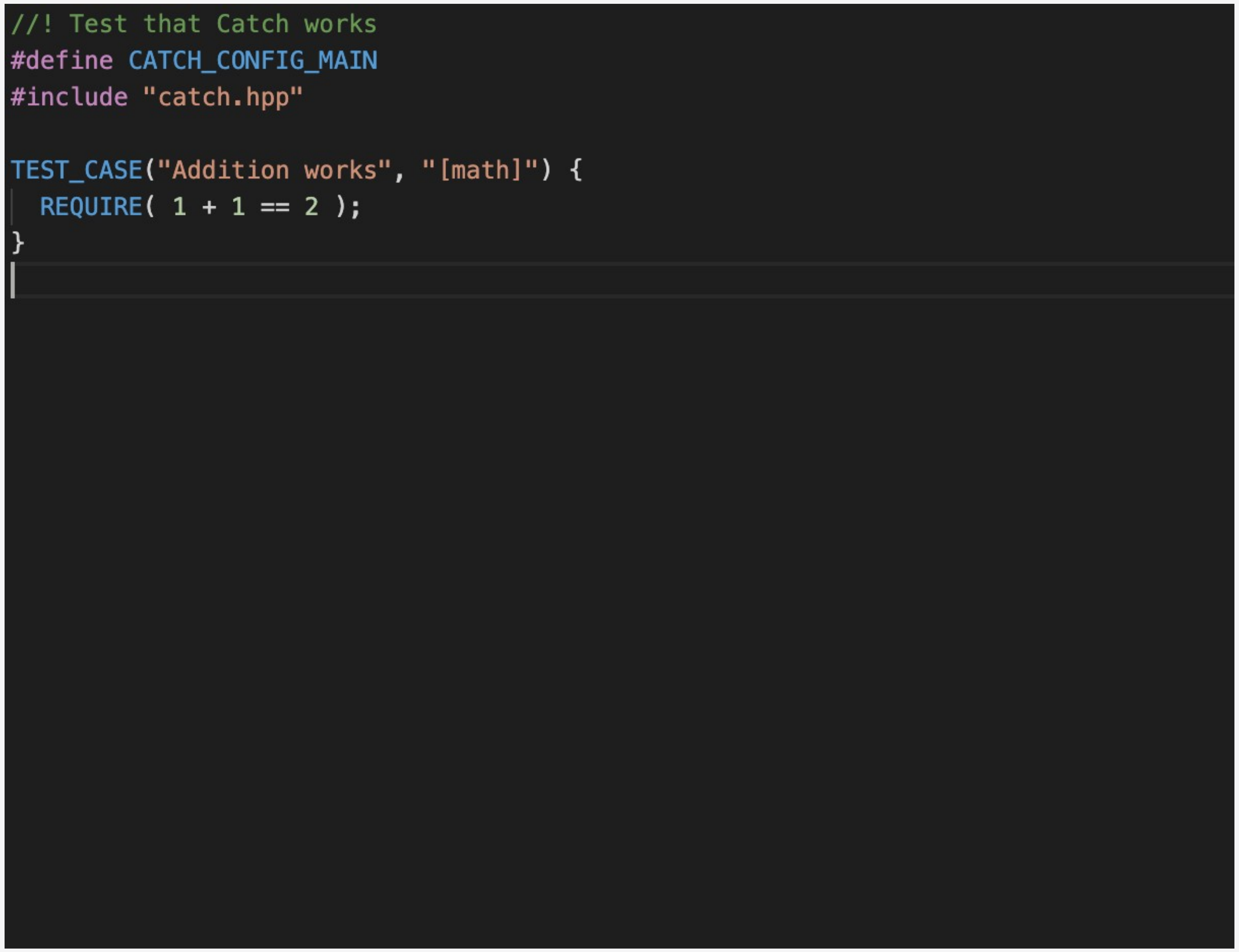
However, macro expansions can be very difficult to debug, so functions should be preferred if possible!

19: Implementing Test Cases

To see how TEST_CASE **and** REQUIRE **work, add the code as shown below to your** testCatch.cpp **file.**

The arguments to TEST_CASE are strings describing the test and "tags" that may be used to group tests (we won't cover these, see the Catch docs for further info).

Inside TEST_CASE, we add a boolean expression inside a REQUIRE call that asserts that the result of 1+1 is 2 (which should be true!!)



```
//! Test that Catch works
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("Addition works", "[math]") {
  REQUIRE( 1 + 1 == 2 );
}
```

Notes

Catch provides several other assertion macros. These can be used for more advanced checks such as comparison of floating point numbers (not as easy as you may think!) and exception throwing.

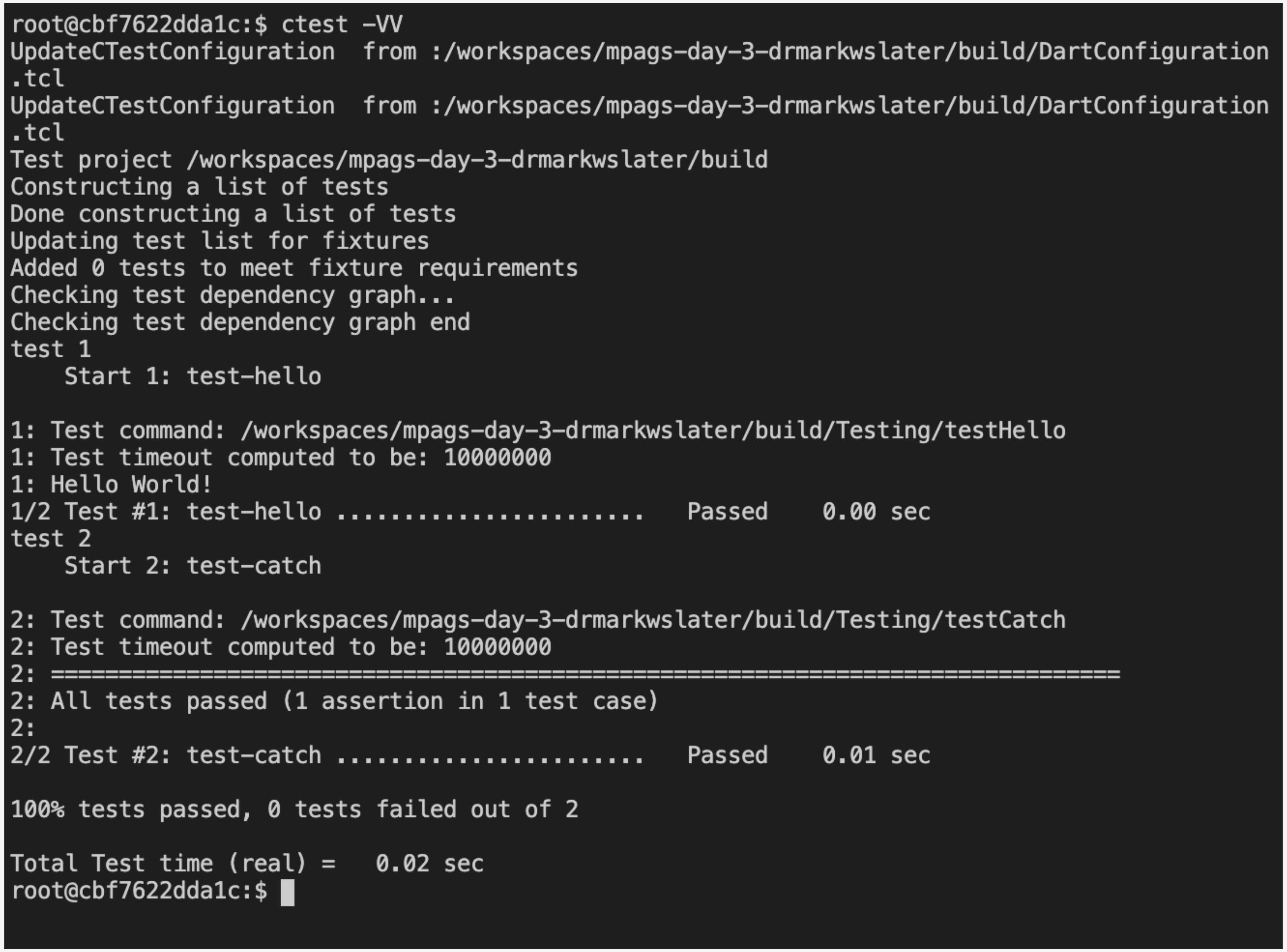
See the reference docs:

https://github.com/catchorg/Catch2/blob/devel/docs/assertions.md

20: Running Test Cases

Save your testCatch.cpp **file, rebuild using** make**, then run** make test **(note that you must run** make **first as** make test **will only run the test program, not rebuild it).**

The test should pass, and we can get more detailed info by running ctest -VV**. Here, Catch tells us about how many test cases and assertions have been run. You can also try running the** testCatch **program directly with the '-s' argument (or use this as an argument to** testCatch **in** add_test**) to see how** REQUIRE **was evaluated.**



```
root@cbf7622dda1c:$ ctest -VV
UpdateCTestConfiguration  from :/workspaces/mpags-day-3-drmarkwslater/build/DartConfiguration
.tcl
UpdateCTestConfiguration  from :/workspaces/mpags-day-3-drmarkwslater/build/DartConfiguration
.tcl
Test project /workspaces/mpags-day-3-drmarkwslater/build
Constructing a list of tests
Done constructing a list of tests
Updating test list for fixtures
Added 0 tests to meet fixture requirements
Checking test dependency graph...
Checking test dependency graph end
test 1
    Start 1: test-hello

1: Test command: /workspaces/mpags-day-3-drmarkwslater/build/Testing/testHello
1: Test timeout computed to be: 10000000
1: Hello World!
1/2 Test #1: test-hello .......................   Passed    0.00 sec
test 2
    Start 2: test-catch

2: Test command: /workspaces/mpags-day-3-drmarkwslater/build/Testing/testCatch
2: Test timeout computed to be: 10000000
2: ===============================================================================
2: All tests passed (1 assertion in 1 test case)
2:
2/2 Test #2: test-catch .......................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.02 sec
root@cbf7622dda1c:$
```

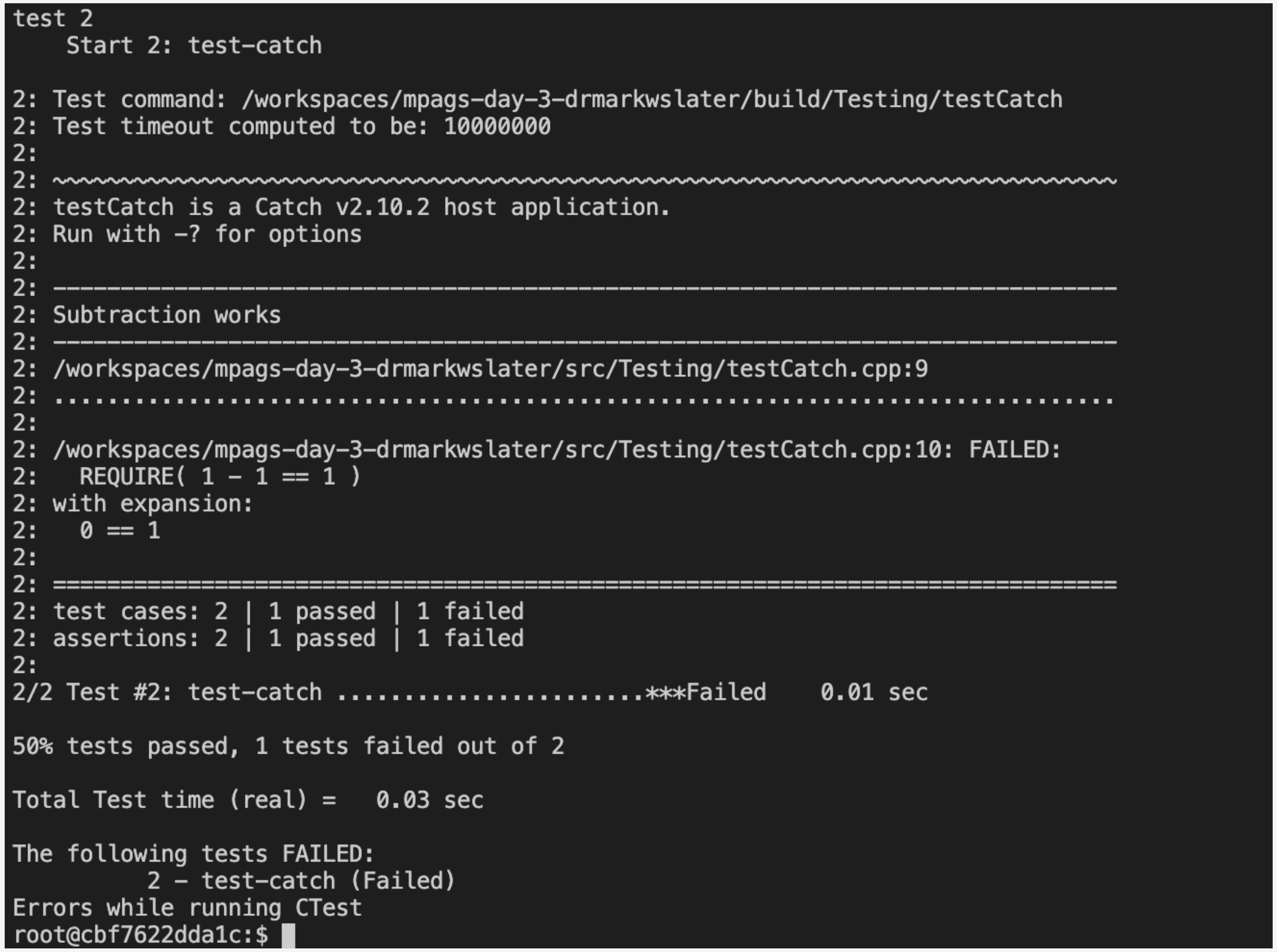
Notes

Generally we just want to run everything a single test executable does. However, we could create one add_test for each test case. The testCatch command would be run every time, but with an argument to select the test case. Catch's "tags" to test cases could also be used here.

21: Failing Tests

To see what happens when tests fail, add an extra TEST_CASE **for subtraction to** testCatch.cpp**, and add a** REQUIRE **using an expression you know will fail (e.g. 1-1 == 1)**

The testCatch program will still compile, but when you run make test, **you should see a failure reported**. Run ctest -VV to get detailed output, and you'll see Catch has **told us which tests failed, and exactly which bit of code caused the failure.**



```
test 2
    Start 2: test-catch

2: Test command: /workspaces/mpags-day-3-drmarkwslater/build/Testing/testCatch
2: Test timeout computed to be: 10000000
2:
2: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2: testCatch is a Catch v2.10.2 host application.
2: Run with -? for options
2:
2: -------------------------------------------------------------------------------
2: Subtraction works
2: -------------------------------------------------------------------------------
2: /workspaces/mpags-day-3-drmarkwslater/src/Testing/testCatch.cpp:9
2: ...............................................................................
2:
2: /workspaces/mpags-day-3-drmarkwslater/src/Testing/testCatch.cpp:10: FAILED:
2:   REQUIRE( 1 - 1 == 1 )
2: with expansion:
2:   0 == 1
2:
2: ===============================================================================
2: test cases: 2 | 1 passed | 1 failed
2: assertions: 2 | 1 passed | 1 failed
2:
2/2 Test #2: test-catch .......................***Failed    0.01 sec

50% tests passed, 1 tests failed out of 2

Total Test time (real) =   0.03 sec

The following tests FAILED:
          2 - test-catch (Failed)
Errors while running CTest
root@cbf7622dda1c:$
```

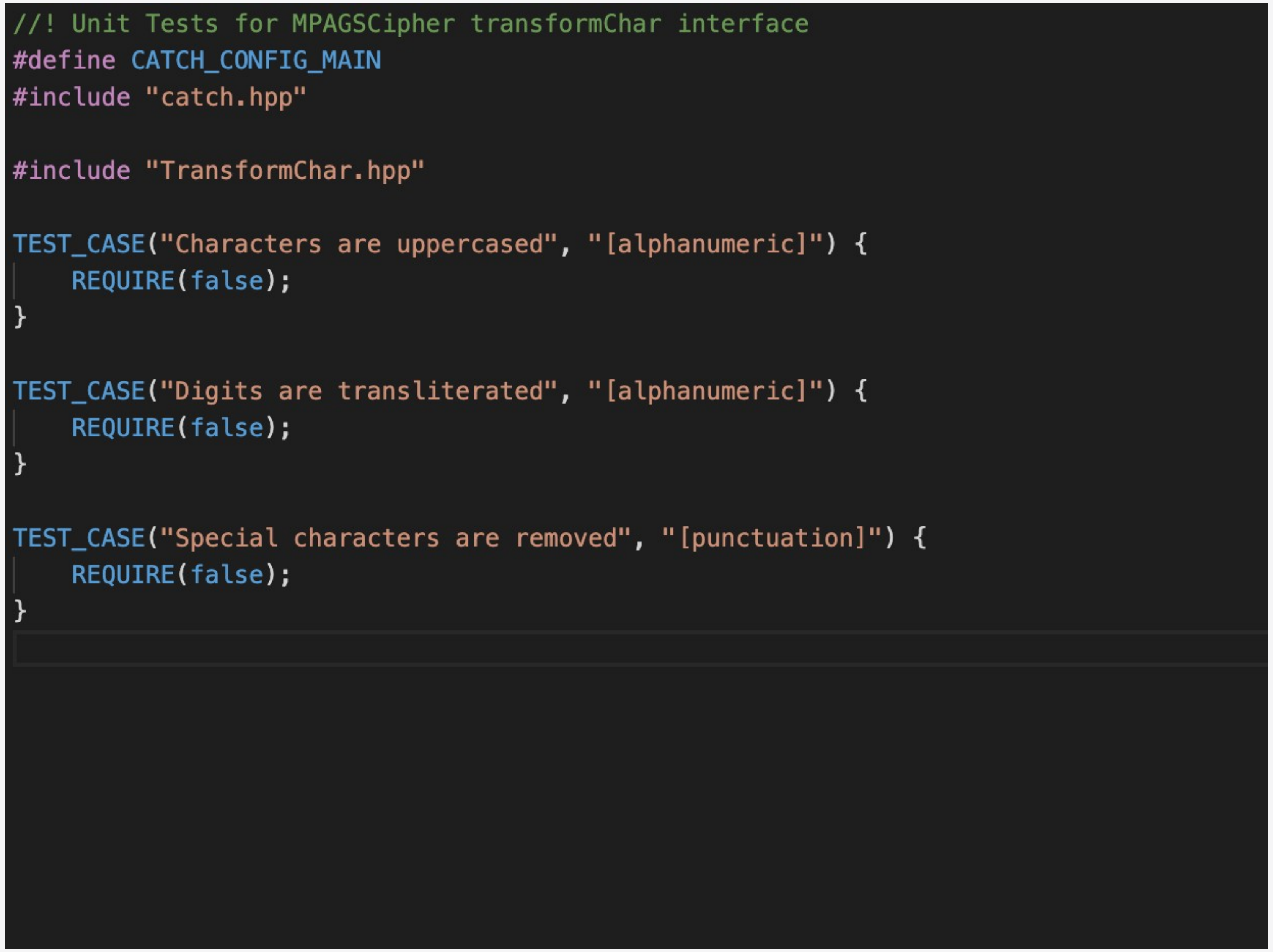
Notes

This output from Catch should give you everything you need to start looking for the problem in the code.

As noted earlier, we'll generally run make test with every rebuild, and then use ctest -VV when we have failing tests to get this extra information.

22: Testing MPAGSCipher with Catch

Before starting to write unit tests for MPAGSCipher, it's worth spending a little time thinking about how to structure them and what to test. The "unit" in unit testing means that ideally we should have one test program per interface (i.e. header file). In that program, test cases can be organised by task or area - for ciphers we might have one test case for encryption, one for decryption. For what to test, we can use our requirements as a starting point and sketch these in as failing tests. We then work through implementing them, from simplest to most complex



```cpp
//! Unit Tests for MPAGSCipher transformChar interface
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include "TransformChar.hpp"

TEST_CASE("Characters are uppercased", "[alphanumeric]") {
    REQUIRE(false);
}

TEST_CASE("Digits are transliterated", "[alphanumeric]") {
    REQUIRE(false);
}

TEST_CASE("Special characters are removed", "[punctuation]") {
    REQUIRE(false);
}
```

Notes

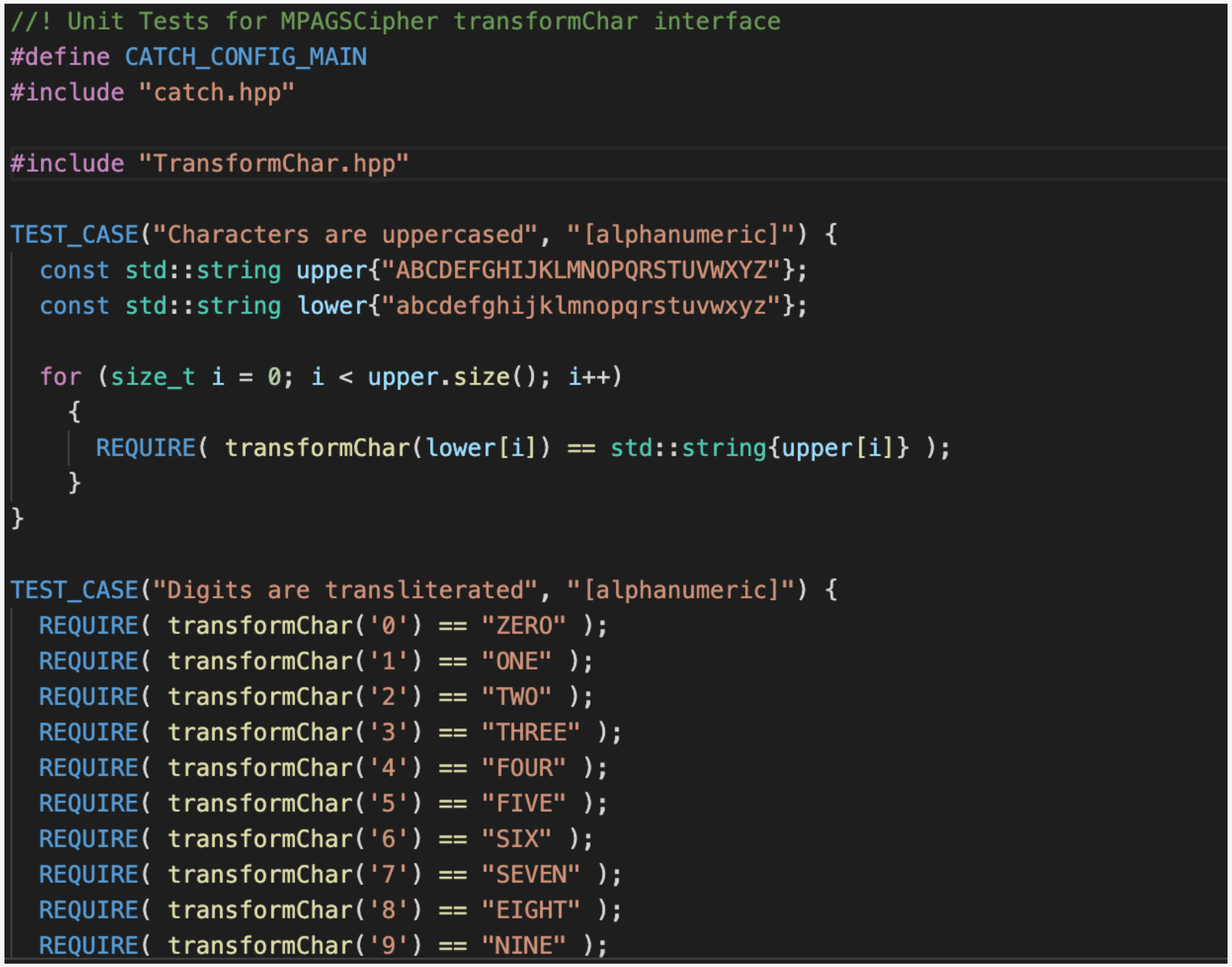
Why write failing tests? It's a note to ourselves that we need a test here, so it failing is a good marker of "needs fixing". This also means we can concentrate on one test at a time.

This is a slightly less upfront version of Test Driven Development – writing tests first and then writing the functionality afterwards to make the tests pass

23: Writing MPAGSCipher Tests

To use, and hence test, MPAGSCipher functions/objects with Catch, we simply #include the relevant header after setting up Catch.

The functions/objects can then be used in test cases and assertions just as they were in other code.



```cpp
//! Unit Tests for MPAGSCipher transformChar interface
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include "TransformChar.hpp"

TEST_CASE("Characters are uppercased", "[alphanumeric]") {
  const std::string upper{"ABCDEFGHIJKLMNOPQRSTUVWXYZ"};
  const std::string lower{"abcdefghijklmnopqrstuvwxyz"};

  for (size_t i = 0; i < upper.size(); i++)
    {
      REQUIRE( transformChar(lower[i]) == std::string{upper[i]} );
    }
}

TEST_CASE("Digits are transliterated", "[alphanumeric]") {
  REQUIRE( transformChar('0') == "ZERO" );
  REQUIRE( transformChar('1') == "ONE" );
  REQUIRE( transformChar('2') == "TWO" );
  REQUIRE( transformChar('3') == "THREE" );
  REQUIRE( transformChar('4') == "FOUR" );
  REQUIRE( transformChar('5') == "FIVE" );
  REQUIRE( transformChar('6') == "SIX" );
  REQUIRE( transformChar('7') == "SEVEN" );
  REQUIRE( transformChar('8') == "EIGHT" );
  REQUIRE( transformChar('9') == "NINE" );
```

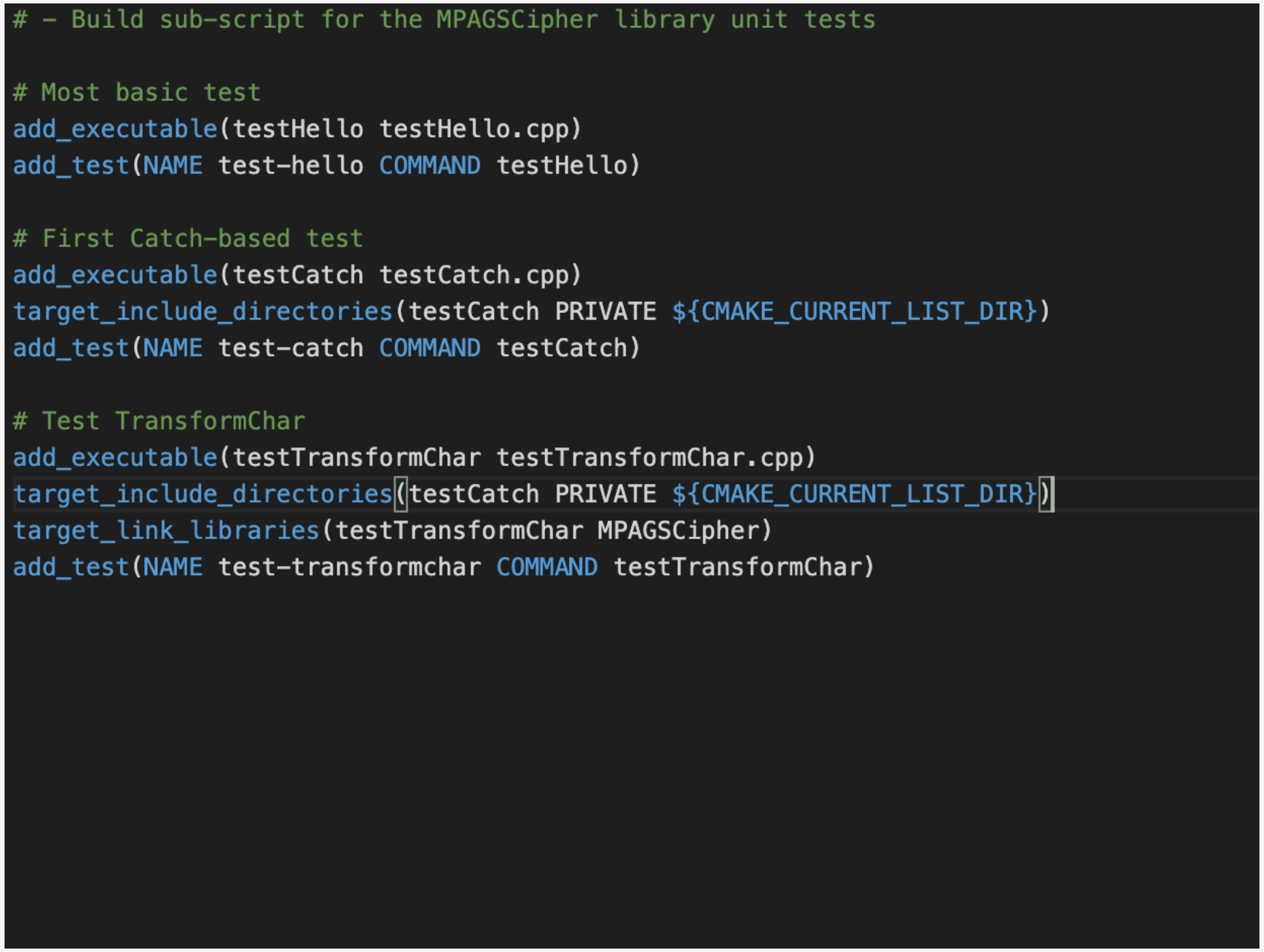
Notes

If you need additional headers, e.g. for C++ Standard Library, to assist in the testing, these can also be included.

24: Building MPAGSCipher Tests

To build the testing program, we build it in CMake just as we did for the basic testCatch program, **but in this case we also need to use** target_link_libraries **to link it to the** MPAGSCipher **library.**

Try building and running your basic MPAGSCipher test program (start with one to begin with) ensuring it can use the relevant MPAGSCipher header and is linked correctly to the static library



```
# - Build sub-script for the MPAGSCipher library unit tests

# Most basic test
add_executable(testHello testHello.cpp)
add_test(NAME test-hello COMMAND testHello)

# First Catch-based test
add_executable(testCatch testCatch.cpp)
target_include_directories(testCatch PRIVATE ${CMAKE_CURRENT_LIST_DIR})
add_test(NAME test-catch COMMAND testCatch)

# Test TransformChar
add_executable(testTransformChar testTransformChar.cpp)
target_include_directories(testCatch PRIVATE ${CMAKE_CURRENT_LIST_DIR})
target_link_libraries(testTransformChar MPAGSCipher)
add_test(NAME test-transformchar COMMAND testTransformChar)
```

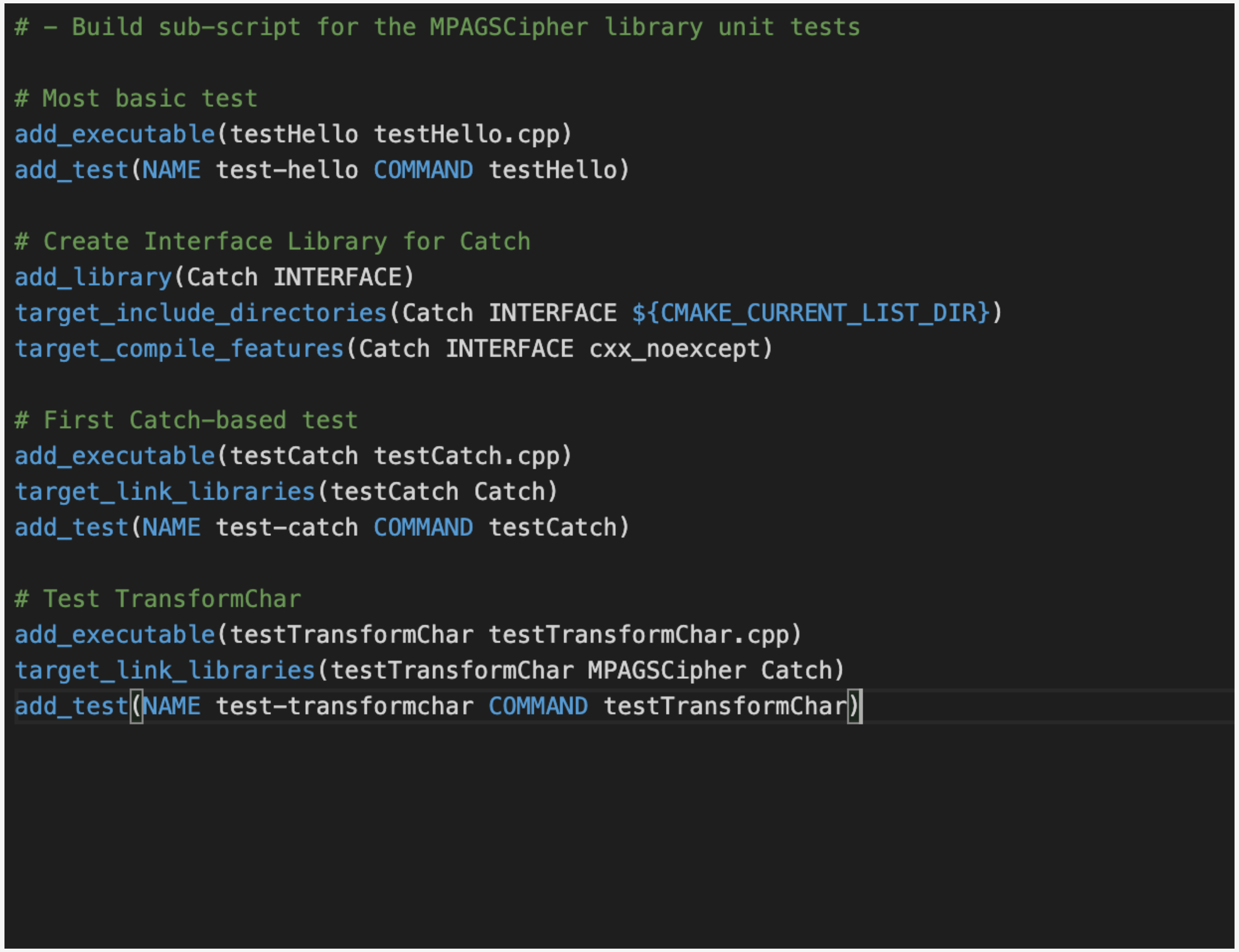
Notes

Now we see the advantage of using the library. In the example on the left, without the library, we would have had to compile transformChar.cpp again for the test program.

We'll see how to reduce the amount of CMake commands in an upcoming slide

25: Simplifying use of Catch.hpp

As further test programs are added, target_include_directories will need to be set for each one so that Catch.hpp is found. We can simplify this through a CMake construct known as an *"interface library"*. This is a "library" with no compiled sources (i.e. has headers only, like Catch or the Eigen Linear Algebra library) but which can have properties like include directories. Users of the library simply "link" to it using target_link_libraries to pick up, here, include directories just like they would when linking to a binary library like MPAGSCipher.



```
# - Build sub-script for the MPAGSCipher library unit tests

# Most basic test
add_executable(testHello testHello.cpp)
add_test(NAME test-hello COMMAND testHello)

# Create Interface Library for Catch
add_library(Catch INTERFACE)
target_include_directories(Catch INTERFACE ${CMAKE_CURRENT_LIST_DIR})
target_compile_features(Catch INTERFACE cxx_noexcept)

# First Catch-based test
add_executable(testCatch testCatch.cpp)
target_link_libraries(testCatch Catch)
add_test(NAME test-catch COMMAND testCatch)

# Test TransformChar
add_executable(testTransformChar testTransformChar.cpp)
target_link_libraries(testTransformChar MPAGSCipher Catch)
add_test(NAME test-transformchar COMMAND testTransformChar)
```

Try This

Use

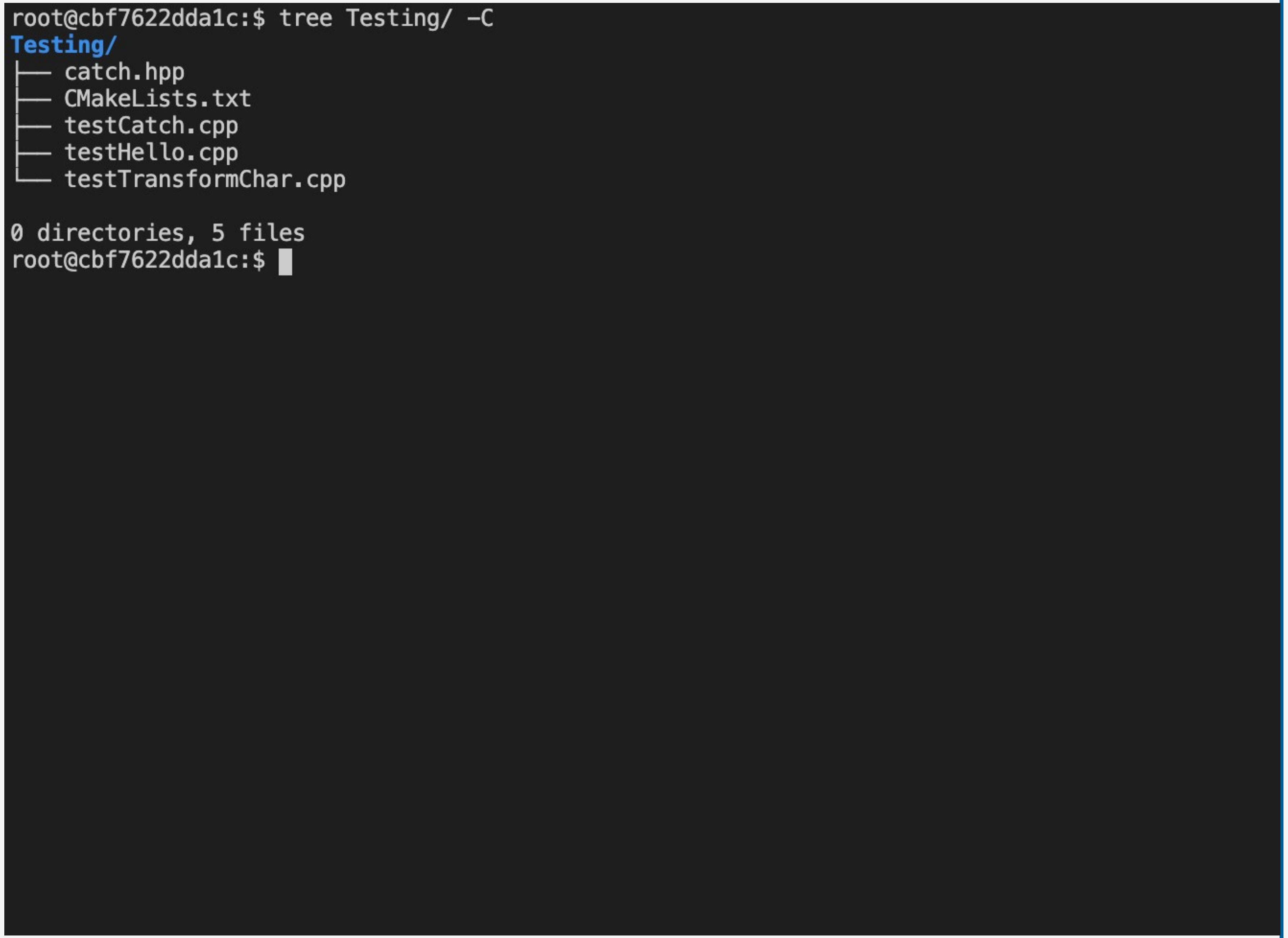
add_library(Catch INTERFACE)

and set its include directories using the `INTERFACE` scope so that your test executables can just link to it to pick up Catch.hpp **correctly.**

26: Adding Further Test Cases and Tests

We've covered the basics of writing tests using Catch, building them with CMake and running with CTest.

Review these and your code and think about other test cases and tests the functions and now classes might need. If you identify one, implement the test and see what happens!



```
root@cbf7622dda1c:$ tree Testing/ -C
Testing/
├── catch.hpp
├── CMakeLists.txt
├── testCatch.cpp
├── testHello.cpp
└── testTransformChar.cpp

0 directories, 5 files
root@cbf7622dda1c:$
```

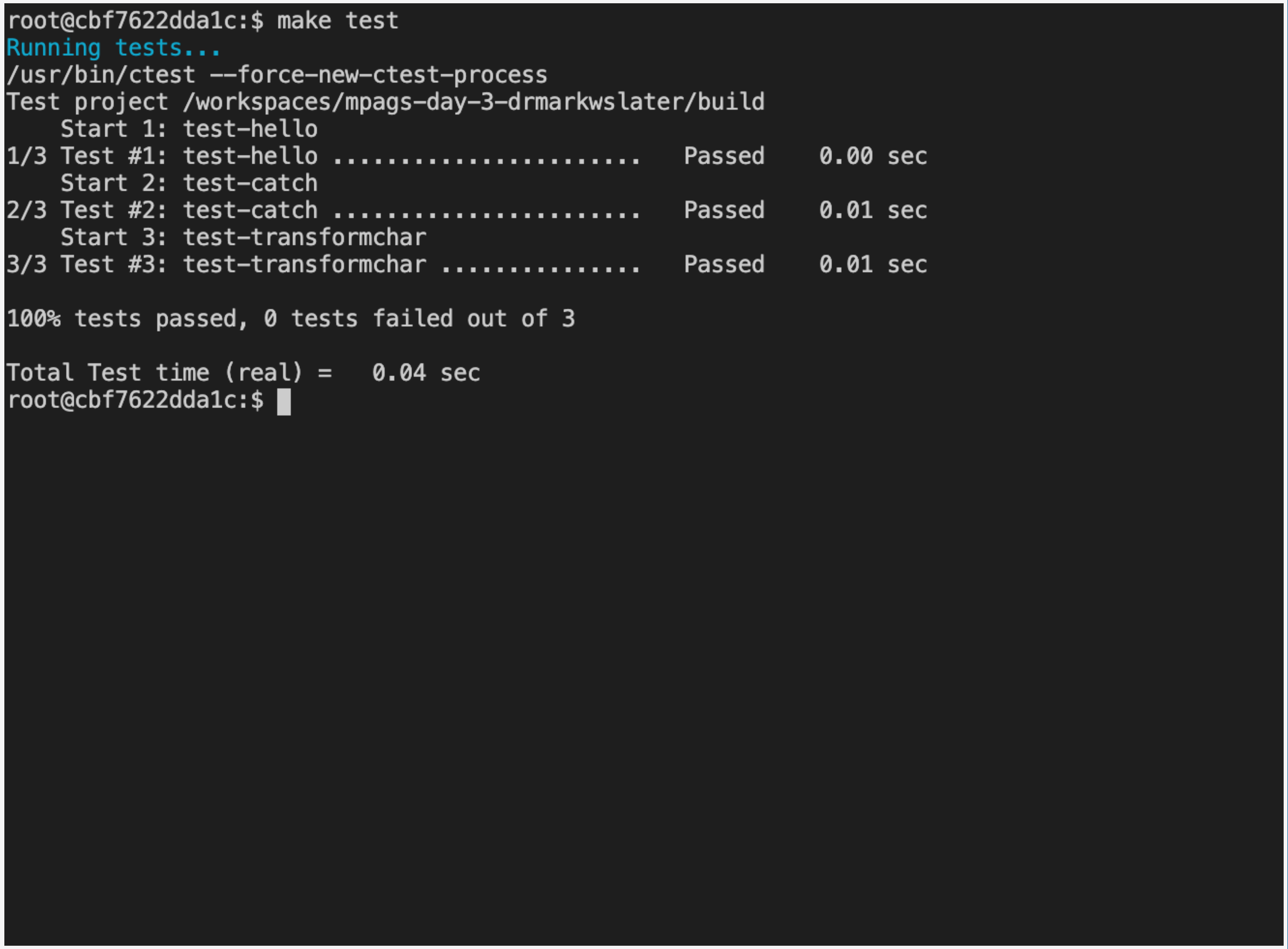
Notes

Generally, it's easiest to have one test executable per unit of functionality (e.g. one test executable for each header). Each executable can have as many test cases as needed.

27: Walkthrough Summary

We've partitioned the build of mpags-cipher into a main program linked to a static implementation library. This has enabled both a simplification of the CMake scripts and set up the implementation to be tested easily by creating test executables that link to the library.

Unit testing has been introduced using the simple Catch framework. We've seen how test programs can be implemented using Catch, built with CMake and run using CTest. Test outputs have been reviewed to see how success and failure cases are reported.



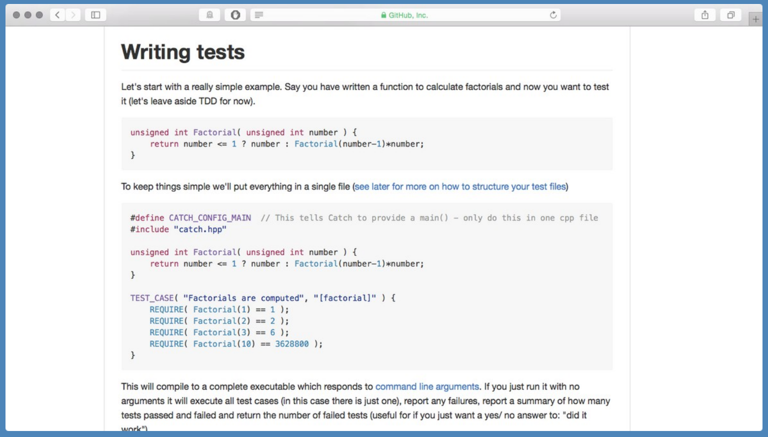
```
root@cbf7622dda1c:$ make test
Running tests...
/usr/bin/ctest --force-new-ctest-process
Test project /workspaces/mpags-day-3-drmarkwslater/build
    Start 1: test-hello
1/3 Test #1: test-hello .......................   Passed    0.00 sec
    Start 2: test-catch
2/3 Test #2: test-catch .......................   Passed    0.01 sec
    Start 3: test-transformchar
3/3 Test #3: test-transformchar ...............   Passed    0.01 sec

100% tests passed, 0 tests failed out of 3

Total Test time (real) =   0.04 sec
root@cbf7622dda1c:$
```

Further Reading

CMake

CMake Documentation

Catch Documentation



GitHub, Inc.

Writing tests

Let's start with a really simple example. Say you have written a function to calculate factorials and now you want to test it (let's leave aside TDD for now).

```
unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}
```

To keep things simple we'll put everything in a single file (see later for more on how to structure your test files)

```
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main() - only do this in one cpp file
#include "catch.hpp"

unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

This will compile to a complete executable which responds to command line arguments. If you just run it with no arguments it will execute all test cases (in this case there is just one), report any failures, report a summary of how many tests passed and failed and return the number of failed tests (useful for if you just want a yes/ no answer to: "did it work")

Homework: Adding Tests for MPAGSCipher Classes

- As you work through your code, determine suitable unit tests for the functionality of each part.

- Implement these tests using Catch, TEST_CASE and REQUIRE as needed. Also look at Catch's SECTION macro as this could help with the Caesar Cipher testing:
  - https://github.com/catchorg/Catch2/blob/devel/docs/tutorial.md

- Build the tests with CMake and ensure they compile and then pass without failure.