Error handling II: Unit testing

Raul P. Pelaez

September 22, 2025

Contents

1 Introduction

The difference between a program and a useful product come from documentation and testing. Loosely speaking, the cost of a product is said to be three times as high as the cost of a program. This is part of the reason why you hear those stories of "a couple persons in a garage creating a billion-dollar software company". The program is the easy part, making that program into a product that can grow, be understood and reused by others and is robust and reliable is the hard part.

Vibe coding, as we understand it today, is a big offender in this sense. Using LLMs, it is straight forward to generate a convincing minimum viable product. However, when it comes to transform it into a real product, ensuring its correctness, reliability, maintainability and usability, the lack of tests and documentation becomes a big problem. A problem that the LLM cannot solve for you, and that you will have a hard time solving, since your cognition of the project will be shallow.

Writing tests and documentation is a boring chore, but one you should simply consider a fact of life and never skip. From now on, every project you start should have a folder structure similar to this:

```
.
+-- CMakeLists.txt
+-- docs\ # A tale for another day
+-- tests\
|   +-- CMakeLists.txt
|   +-- test_component1.c
|   \-- ...
+-- environment.yml
\-- src\
    +-- CMakeLists.txt
    \-- ...
```

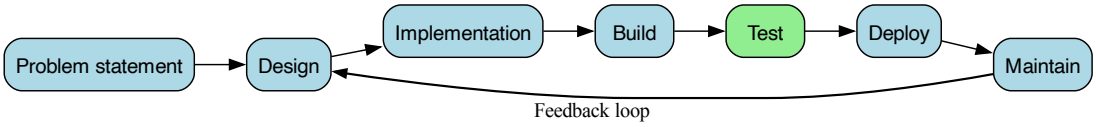
The specifics will change depending on the language(s) your project is written, but the gist of it should remain. For instance, in a pure Python project, you would have a `pyproject.toml` file instead of a `CMakeLists.txt` file. It is also ok if you stray a bit away from this structure. For example, you might have a complex folder structure and decide to integrate the test files alongside the tested components. CMake can be easily configured anyhow.

Advice

Do not neglect software engineering
Testing and documentation take discipline, but they are a tremendous investment into the future of your project and will greatly reduce the long-term cost of developing and maintaining it.

2 Testing

Today, we will focus on testing, an important part of the software development cycle:



There are many types of tests, some of the most common ones are:

- **Unit tests**: Test individual components or functions in isolation. They ensure that each part of the code works as intended.
- **Integration tests**: Verify that different components work together correctly. They test the interactions between modules. e.g. API tests (frontend-backend communication in a web app).
- **Functional tests**: Validate that the software meets the specified requirements and behaves as expected from an end-user perspective. e.g. UI tests.
- **Performance tests**: Assess the speed, responsiveness, and scalability of the software under various conditions.

Integration and functional tests go beyond the scope of this programming course. You will, with some luck, learn about them in a future DevOps course. Today, we will focus on unit tests. IMHO, they are the most important type of tests, and the ones that will give you the most bang for your buck.

Info

Unit tests
A small, isolated piece of code that verifies the correctness of a specific function or *component* in your software. They are typically written by developers during the development process to ensure that individual units of code work as intended. A component might be a single function, a module, a small internal library, or the entire application. Unit tests work best by thinking about them in a hierarchical manner, where you start testing the smallest components and then build up to larger ones.

Info

Test-Driven Development (TDD)
A software development approach where tests are written before the actual code. The process typically follows these steps:

1. Write a test for a new feature or functionality.
2. Write the minimum amount of code needed to make the test pass.
3. Repeat the process.

In TDD, one writes the tests **before** the code itself. This forces the developer to design the interface of a component before implementing it, leading to better-designed and more testable code.

Advice

Be pragmatic
Take a pragmatic approach to software development. Be skeptic of methodologies that promise to solve all your problems. Reject dogmatic principles or rules. Ran away from self-proclaimed "gurus" that claim to have the one true way of doing things. Everything in tech is a trade-off, there is no definitive strategy.
For instance, the "correct" application of TDD, as described by the authors in [1], explain that when they say "the minimum amount of code needed to make the test pass", they really mean it. An actual example they give is along this line, in pseudo code. Imagine a function that is supposed to multiply numbers, we could start by writing a test for it like:

```
def test_mult():
    assert my_mult(3,4) == 12
```

The minimum code that satisfies this test is:

```
def my_mult(a,b):
    return 12
```

The next step would be to add some more checks to the test (it is important that we never remove tests), which would force us to generalize our function, and so on and so forth.
It is, IMHO, ok to get a bit ahead when the specifications are clear and skip a few iterations. Again, be pragmatic, try things out and stick with what works for you and what the community accepts as a standard.
If you like writing code and building software, I strongly recommend you read The Pragmatic Programmer [2].

The structure of a Unit test is, regardless of the language or framework used, pretty similar. Multiple libraries exist to help reduce the boilerplate, or common code, needed to write tests. We typically want to use one of these libraries, since they talk nicely and automagically with the rest of the tooling, like CMake and VSCode.

The most popular testing framework for C/C++ is GTest (Google Test). It is quite easy to integrate into a CMake project (a few lines in the `CMakeLists.txt` will download it and make it available to your tests). It is also very easy to use, open source and well documented. It is, however, a C++ library. While that is not really a problem, IMO it introduces a bit of unnecessary friction for this particular course. Instead, we will be using utest.h, a single-header C library that closely mimics the GTest API. In the future, should you need to do it, you will easily be able to switch to GTest.

Integrating utest.h amounts to downloading the `utest.h` file from the repository and placing it somewhere in your project. The course repository already has it under `third_party/utest.h`. You can then include it in your test files and start writing tests.

Let us learn about Unit testing by example.

2.1 How a Unit test looks like

A unit test is some code that we can execute and that will, in essence, report a pass or a fail. In practice, we usually group tests into test suites, which are collections of functions with assertions that check the correctness of a component. A test suite can contain multiple tests, and each test can contain multiple assertions. How we organize our test functions into different source files and test suites is not very important. As we will see, as long as we register the tests with CTest, CMake's testing tool, they will all be discovered and run automatically.

Here is a simple example of a test file using utest.h. Lets test something trivial, like the C standard library function `strlen()`, which computes the length of a string. Create a file called `test_strlen.c` and add the following content:

```c
#include "utest.h"
#include <string.h> // For strlen()
UTEST(strlen, basic) {
  ASSERT_EQ(strlen(""), 0);
  ASSERT_EQ(strlen("a"), 1);
  ASSERT_EQ(strlen("hello"), 5);
```

```
}
UTEST(strlen, spaces){
  ASSERT_EQ(strlen("hello world"), 11);
  ASSERT_EQ(strlen("hello\tworld"), 11);
  ASSERT_EQ(strlen("hello\nworld"), 11);
}

UTEST_MAIN()
```

The `UTEST` macro registers a new test function. The first argument is the name of the test suite, and the second argument is the name of the test inside the suite. The body of the test function contains assertions that check the correctness of the code being tested. The utest.h library provides multiple assertion macros, like `ASSERT_EQ`, `ASSERT_NE`, `ASSERT_TRUE`, `ASSERT_FALSE`, etc. You can find the full list in its documentation. If you check the documentation for GTest, you will see that it is essentially the same API.

Finally, we need to call the `UTEST_MAIN()` macro, which defines the `main()` function that will run all the tests.

To compile and run the test, we can integrate it into a CMake project (we will see how in a moment). But for simplicity, we can just compile and run it manually with:

```
  $ cc -o test_strlen test_strlen.c
  $ ./test_strlen
  [==========] Running 2 test cases.
[ RUN      ] strlen.basic
[       OK ] strlen.basic (42ns)
[ RUN      ] strlen.spaces
[       OK ] strlen.spaces (0ns)
[==========] 2 test cases ran.
[  PASSED  ] 2 tests.
```

Make sure `utest.h` is in the same folder as `test_strlen.c`.

You should see a something similar to the above, i.e. a detailed report of the tests that were run and their results. Try to break one test by changing an expected value, and see how the output changes.

If we are careful and disciplined with the names and scopes of our tests, this report will help us quickly identify what is broken when something goes wrong. The debugging power of a good test suite is immense, imagine that your project has hundreds of files and thousands of tests (this is not uncommon at all, see TorchMD-NET's test folder for an example). This concise report will list all failing tests, giving you an eagle-eye view of what is broken, e.g. "all the failing tests are using the newtonian integrator, so the problem must be there".

2.2 CMake integration

One of the main advantages of unit tests is that once we have written one, it will stay there forever. That means that when we implement component B, we will be able to check that component A has not been broken in the process.

Insight

The absence of this kind of regression tests that we get "for free" with unit tests is one of the main reasons why software tends to rot over time. As new features are added, old ones break, and since there are no tests to catch these regressions, they go unnoticed until they cause a problem in production.

There is a catch, however. Unit tests are only useful if we constantly run them. If we have to manually compile and run each test, we will quickly get tired of it and stop doing it. This is where CMake comes in. We can register executables as tests for CMake's testing tool, `ctest`, and then run all tests with a single command.

Advanced

Continuous Testing
Even if tests are integrated into CMake, we still have to remember to call `ctest` manually after every change. In a professional setting you would go a step futher and integrate the tests into a Continuous Integration (CI) system, like GitHub Actions, GitLab CI, Jenkins, etc. This way, the tests will be run automatically every time you push a change to the repository.

Lets see how to do this with the `test_strlen.c` example above. Create a `CMakeLists.txt` file in the same folder as `test_strlen.c` with the following content:

```
cmake_minimum_required(VERSION 3.21) # Always required in a CMakeLists
project(TestStrlen C) # Name of the project and main language
add_executable(test_strlen test_strlen.c) # Create an executable from the source file
add_test(NAME test_strlen COMMAND test_strlen) # Register the test with CTest
```

Then, from the terminal, run:

```
$ cmake -B build
$ cmake --build build
$ ctest --test-dir build
```

You should see the same output as before, but now we can run the tests with a single command. You can also run `ctest` with the `-V` flag to get a more verbose output. Naturally, you can add more test files and register them in the same way, and `ctest` will run them all.

On the other hand, we typically separate the tests from the source code of the project. This is why we usually have a `tests/` folder in the root of the project, with its own `CMakeLists.txt` file. The root `CMakeLists.txt` file will then delegate to the `tests/CMakeLists.txt` file. We will see how to do this in the exercise below.

2.3 VSCode integration

VSCode has built-in support for CMake and CTest. If you have the CMake Tools extension installed, you can easily configure, build and run tests from the VSCode interface.

Advice

In general, it is a good idea to say yes when VSCode recommends you to install an extension.

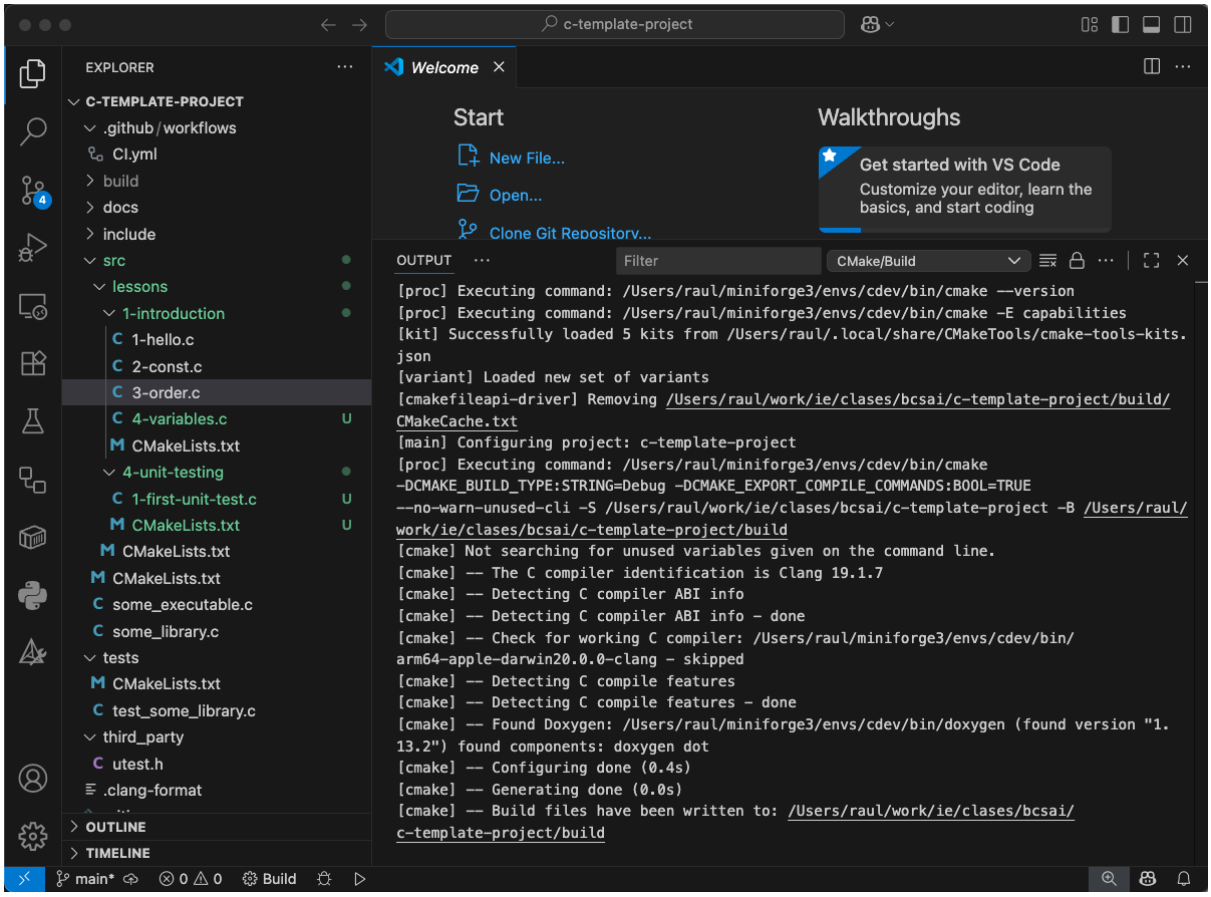
As usual, open the terminal and navigate (using the `cd` command) to the root of your project. Then, activate the conda environment for the course (if you have not done so already) and open VSCode in the current folder with:

```
$ conda activate cdev # or the name of your environment
$ code .
```

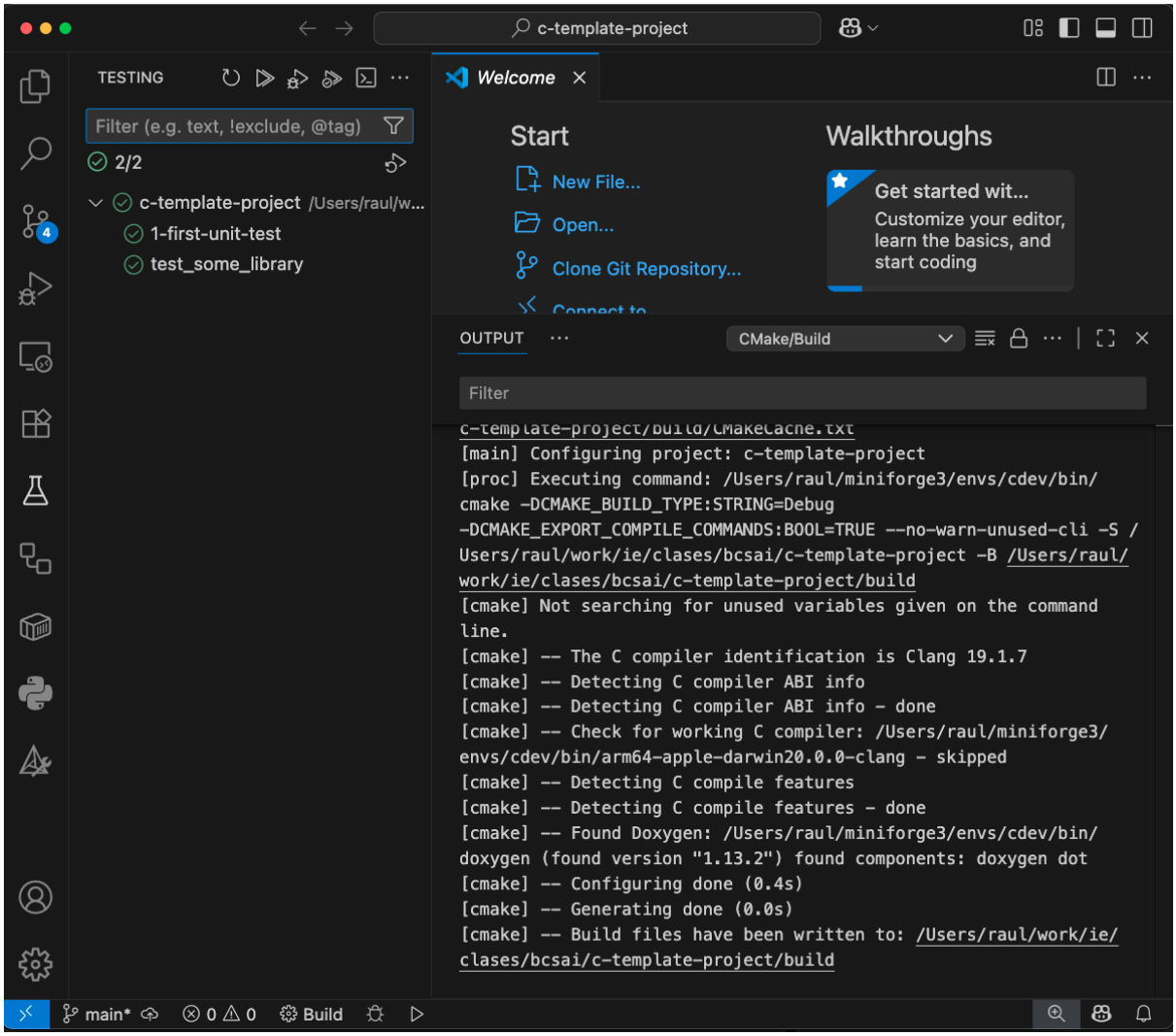
It would be a good idea to copy the `environment.yml` file from the course repository into the root of your project, so that you can easily recreate the environment in the future.

It is important that you open VSCode like this, since it will inherit the conda environment from the terminal. If you open VSCode by clicking on its icon, it will not have access to the conda environment, and you will have to configure it manually.

Once VSCode is open, you should be presented with something like this:



VSCode picked up the CMake files automatically and built them, placing the results in the newly created `build` directory (we can see this in the OUTPUT terminal). VSCode also realized that our CMake includes tests, so the testing tab appear in the left sidebar. If you click on it, you will see the tests that were discovered by CMake:



The tab shows all tests that were discovered by CMake. You can run all tests by clicking on the "Run All Tests" button, or you can run individual tests by clicking on the play button next to each test. You can also see the output of each test by clicking on the test name. VSCode will mark the tests that pass with a green check mark, and the tests that fail with a red cross.

Insight

I personally never use this integration. I rely on more automated workflows, such as CI/CD pipelines, to run the tests for me. Even when I am developing locally, I find it much more convenient and efficient to build the project and run the tests from the terminal.
What I will usually do is to have a terminal open in which I run `cmake --build build && ctest -V --test-dir build`. Every time I want to make sure tests are passing, I `Alt-TAB` to that terminal, press the up arrow to recall the last command, and press `Enter` to run it again. This way, I can run the tests and see the output with just a few keyboard chords.

3 Exercises

Goal

Your first project with tests
Let us create a simple C project that asks for the users name and returns the length of the name. We will create a full-fledged project, that uses CMake for building, and that has a test suite to ensure the correctness of the code.

hint

- Use the course template project repository to guide you. It is available at `https://github.com/RaulPPelaez/c-template-project`.

Learning goal

- I am aware that the functionality of this exercise is not very exciting, but I want you to focus on the truly complicated part of the course thus far: setting up a project with CMake and writing tests.
- I promise to give you the chance to code some cool stuff soon :).
- I am also using this exercise as an excuse to force you into getting familiar with the template repository, since it is a toned down version of the kind of proffessional project you will be working on in the future.

I will guide you step by step through the process. As lessons go by, I will expect grab your hand less and less.

Milestone

Start a new project by navigating to a folder of your choice and opening VSCode there:

```
$ cd ~ # go home
$ cd path/to/your/course/folder
$ mkdir lesson4
$ cd lesson4
$ code .
```

Milestone

Copy the environment file from the repo and place it in the root of your project. Then, create and activate the conda environment (do this from a terminal inside VSCode, which you can open with `Command+J` on MacOS or `Ctrl+J` on Windows/Linux):

```
$ conda env update -n lesson4
$ conda activate lesson4
```

Milestone

Using the terminal inside VSCode, create the folder structure of your project:

```
$ mkdir src tests include third_party
```

Copy the `utest.h` file from the course repository into the `third_party` folder of your project. The `src` folder will contain the source code of your project, the `include` folder will contain the headers (the public API), and the `tests` folder will contain the tests.

Insight

Separating the headers from the source code is a good practice, since it allows you to clearly separate the public API from the implementation. It also makes it easier for the tests to include the headers, since they will not need to know about the internal structure of the source code.

Milestone

From the terminal inside VSCode, create the `CMakeLists.txt` files in the root of your project. You can do so by running:

```
$ code CMakeLists.txt
```

Add the following content to the `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 3.21) # Always required in a CMakeLists
project(Homework C) # Name of the project and main language
# Makes sources in this folder available for all sources to "include"
include_directories(${PROJECT_SOURCE_DIR}/include)
add_subdirectory(src) # Delegate to src/CMakeLists.txt
enable_testing() # Enable the testing integration
add_subdirectory(tests) # Delegate to tests/CMakeLists.txt
```

Insight

I added comments to the snippet above to explain to you what each line does for didactic purposes. It is in general not a good idea to comment every line of your code with such obvious and repetitive comments (you can ask me why). I tell you this because ChatGPT **loves** to do this, and it is such a bad practice!

Milestone

Write a prototype for the function that will compute the length of a string. Create a file called `name_length.c` inside the `src` folder:

```
$ code src/name_length.c
```

Add the following content to the `src/name_length.c` file:

```
#include "name_length.h"
int name_length(const char* name) {
    return 0;
}
```

Create a header file called `name_length.h` inside the `include` folder:

```
$ code include/name_length.h
```

Add the following content to the `include/name_length.h` file:

```
#ifndef NAME_LENGTH_H
#define NAME_LENGTH_H
int name_length(const char* name);
#endif // NAME_LENGTH_H
```

Finally, lets create a `main.c` file inside the `src` folder that will contain the `main()` function of our program:

```
$ code src/main.c
```

Add the following content to the `src/main.c` file:

```
#include <stdio.h>
#include "name_length.h"
  int main() {
      printf("This is just a prototype, calling name_length with a test name:\\n");
      int len = name_length("Raul");
      printf("The length of the name is: %d\\n", len);
      return 0;
  }
```

We are just creating a basic layout for our project.

Insight

Note that we already made some design choices. For instance, we decided that the function that computes the length of a name will be called `name_length` and that it will take a `const char*` as input and return an `int`. In a real project, this is the kind of decision that you would want to think about long and hard before committing to it. Changing the API of a function down the line will have a cost in terms of time and effort. Even now, in this trivial example, changing the return type of the function to a `size_t` will require you to change THREE files! Can you imagine how much more complicated it would be in a real project with hundreds of files and multiple developers?
Note that some decisions we take "unknowingly". For instance, by choosing to return an `int`, we are implicitly limiting the maximum length of a name to the maximum value of an integer.

Milestone

Lets inform CMake about these source files. We created a library called `name_length`, and an executable called `main`. Remember that the root `CMakeLists.txt` file already exposes the `include` folder to all subdirectories. Create the `CMakeLists.txt` file inside the `src` folder:

```
$ code src/CMakeLists.txt
```

Add the following content to the `src/CMakeLists.txt` file:

```
add_library(name_length name_length.c) # Create a library from the source file
add_executable(main main.c) # Create an executable from the source file
target_link_libraries(main name_length) # Link the executable to the library
```

Milestone

At this stage, you should be able to build and run the program. From the terminal inside VSCode, run:

```
$ cmake -B build
$ cmake --build build
```

This will create a `build` folder in the root of your project, containing all the build files. You can then run the program with:

```
$ ./build/src/main
```

Make sure it runs and prints the expected output before mobing on.

Milestone

Let us move on to writing tests. Create a `CMakeLists.txt` file inside the `tests` folder:

```
$ code tests/CMakeLists.txt
```

Add the following content to the `tests/CMakeLists.txt` file:

```
# Create an executable from the source file
add_executable(test_name_length test_name_length.c)
# Link the executable to the library
target_link_libraries(test_name_length name_length)
# Make sure the test can find the headers
target_include_directories(test_name_length PRIVATE ${PROJECT_SOURCE_DIR}/third_party)
# Register the test with CTest
add_test(NAME test_name_length COMMAND test_name_length)
```

Note how we linked with the `name_length` library, even though it was defined in another folder. CMake makes this easy for us.

Place the `utest.h` header file inside the `third_party` folder of your project.

Milestone

Create a test file called `test_name_length.c` inside the `tests` folder:

```
$ code tests/test_name_length.c
```

Add the following content to the `tests/test_name_length.c` file:

```
#include "utest.h"
#include "name_length.h"
UTEST(name_length, basic_tests) {
  ASSERT_EQ(name_length(""), 0);
}
```

At this stage, we should be able to build and run the tests. From the terminal inside VSCode, run:

```
$ cmake -B build
$ cmake --build build
$ ctest --test-dir build
```

This will build the tests and run them. You should see that the test passes.

Insight

This process that we just did scales to projects of arbitrary complexity. Now we just need to add more tests and functionality.

Milestone

Finish the implementation of the `name_length` function by adding more tests and making them pass, give TDD a try. Add this test to the `tests/test_name_length.c` file as a way to check that your implementation is up to par:

```
UTEST(name_length, advanced_tests) {
  ASSERT_EQ(name_length("Raul"), 4);
  // Note that the space does NOT count!
  ASSERT_EQ(name_length("Ana Maria"), 8);
}
```

References

[1] Kent Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003. ISBN 978-0-321-14653-3. Google-Books-ID: CUlsAQAAQBAJ.

[2] D. Thomas and A. Hunt. *The Pragmatic Programmer: Your journey to mastery, 20th Anniversary Edition*. Pearson Education, 2019. ISBN 978-0-13-595691-5. URL `https://books.google.es/books?id=LhOlDwAAQBAJ`.