Testing and documentation

Raul P. Pelaez

October 17, 2024

Contents

1 Introduction

The difference between a program and a useful product is 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.
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:

Source code

```
.
+-- CMakeLists.txt
+-- docs\
|   +-- CMakeLists.txt
|   +-- Doxyfile.in
|   \-- source\
+-- tests\
|   +-- CMakeLists.txt
|   +-- test_component1.cpp
|   \-- ...
+-- 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 setup.py file instead of a CMakeLists.txt file and no Doxyfile.in.

2 Testing

Unit testing is a crucial practice in software development that involves testing individual components or functions of a program in isolation. In C++, Google Test (GTest) is a popular framework for writing and running unit tests. GTest provides a rich set of assertions, test fixtures, and test discovery features that make it easy to create comprehensive test suites. By using GTest, developers can verify that each unit of their code behaves as expected, catch bugs early in the development process, and ensure that changes don't introduce unintended side effects.

Advice

Every component and functionality that is user-facing should have a unit test battery. You should run this battery every time you make a change to the code, even when changes are not related to the component being tested.
Untested code is useless, because you cannot trust it to work as expected. It is also a nightmare to refactor, because you cannot be sure if you are breaking something.

Test-Driven Development (TDD) is a software development approach where tests are written before the actual code. The TDD cycle, often called "Red-Green-Refactor," consists of three main steps:

- Red: Write a failing test that defines a desired improvement or new function.
- Green: Write the minimal amount of code to make the test pass.
- Refactor: Clean up the code while ensuring the tests still pass.

TDD encourages developers to think about the design and requirements of their code before implementation. This approach can lead to more modular, flexible, and maintainable code. It also provides a safety net for future changes, as the comprehensive test suite helps catch regressions quickly. While TDD can initially slow down development, it often results in higher quality code and can save time in the long run by reducing bugs and simplifying debugging.

While I encourage you to give TDD a good try, it is one of those things that you do not want to be dogmatic about. Software development is a highly nuanced process with no one-size-fits-all solutions. Use critical thinking and take away from this philosophy what works for you in each situation.

2.1 GTest

2.1.1 Getting GTest

We can use CMake to get GTest when needed, they integrate well. You can use this construct in the CMakeLists.txt of your test folder to get GTest:

Source code

```
include(FetchContent)
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG v1.15.2
)
# For Windows: Prevent overriding the parent project's compiler/linker settings
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)
include(GoogleTest)
```

Then, whenever you want to add a source code with tests you can do something like this:

Source code

```
add_executable(test_my_library  test_my_library.cpp)
target_link_libraries(test_my_library GTest::gtest_main)
gtest_discover_tests(test_my_library)
```

Note that you will have to adapt this part to the specifics of your project.

CMake comes with a tool called ctest, which can be used to run the tests. You can run all the tests in your project by running this in the build directory:

Source code

```
$ ctest
```

In order to enable testing in CMake, you need to add this line to your root CMakeLists.txt file:

Source code

```
enable_testing()
```

2.1.2 Writing tests

Here is an example of a test using GTest:

Source code

```cpp
//file tests/test_my_library.cpp
#include <gtest/gtest.h>
#include "my_library.h"

using namespace my_library;
TEST(MyLibraryTest, AddTest) {
EXPECT_EQ(add(1, 2), 3);
EXPECT_EQ(add(-1, 1), 0);
}
```

There are many more macros, assertions, and features in GTest that you can use to write comprehensive test suites. But just this simple construct can take you very far.

2.2 Key new concepts

Info

Unit testing
A software testing technique that involves testing individual components or functions of a program in isolation.

Info

Google Test (GTest)
A popular C++ testing framework that provides a rich set of assertions, test fixtures, and test discovery features.

Info

CMake's FetchContent
A CMake module that simplifies the process of downloading and building external projects. It is best used sparingly, only when the dependency specifically lists this as an installation option.

Info

CMake's enable_testing
Adding this anywhere in combination with calls to `gtest_discover_tests` will make CMake generate a target that runs all the tests.

3 Documentation

Python's docstring mechanism is extremely powerful because it incentivises keeping the documentation close to the code. This is really important because if there is one thing worse than no documentation, it is outdated documentation.
While C++ does not offer a built-in system for documentation (like Python's docstrings), its mature ecosystem has come up with many solutions for it.

Doxygen is a tool that can generate documentation from specially formatted comments in the code. It is widely used in the C++ community and it is the tool we will be using in this course. Alas, Doxygen is not known for producing beautiful and user-friendly web pages (see an example). Doxygen is really

good at extracting the comments from the code and storing them in a database. We typically then connect Doxygen with a tool like Sphinx to generate the final documentation.

Sphinx is a tool that can generate documentation from reStructuredText (a markdown language) files. Sphinx is the web-creation library behind most readthedocs pages and provides more "modern" webs than Doxygen by default, see an example.
In order to connect the Doxygen database with Sphinx we use a tool called breathe.

Advanced

Finally, we can use exhale to produce full automatic API reference pages (a page with a list of all classes, functions, etc. in the code).

Advice

This is one of those things that is quite a mess to set up and get your head around the first time you do it. But then you can just copy-paste from a previous project and adapt it to a new one. It is a one-time investment that pays off in the long run.

Books have been written about how to write good documentation (you have some books in the recommended bibliography) and it is one of those skills that takes time to develop. One key concept you should understand is that any functionality, function, class, etc that your code exposes to the user should be documented in a way that is easy to understand without having to look at the implementation.

Advice

You should consider that an undocumented feature is as good as one that does not exist. The user will not know how to use it or even if they should use it ("is this part of the API or some internal function with some undisclosed side-effects?").
There is one particular user that will hate you if you do not document your code: you three months after writing the code.

Warning

Beware of the fallacy of not writing documentation. Common excuses include (counter points in parentheses):

- "The code is self-explanatory" (it is not)
- "I will write it later" (you won't)
- "I don't have time" (better to spend a few hours now than a few months down the line)
- "This is not that big of a project" (it will grow and get out of hand)
- "The interface is not stable yet" (it will never be stable enough)

You would not believe how many times a simple weekend project has become a fundamental keystone in the worlds software infrastructure. Related XKCD. Linux was born as a "hobby that won't be big and professional".

3.1 Key new concepts

Info

Documentation
The process of writing and maintaining information about a software product's features, functionality, and architecture.
In C++, we will use three components to generate documentation:

- Doxygen: Extract comments into a database
- Sphinx: Generate webpages from reStructuredText files
- Breathe: Connect Doxygen with Sphinx

3.2 Doxygen

I recommend you install doxygen using conda. Simply add it to your environment file for a project.
Code-wise, Doxygen amounts to adding comments to your code in a special format. Here is an example of a function with Doxygen comments:

Source code

```
/**
 * @brief A function that adds two numbers.
 * @param a The first number.
 * @param b The second number.
 * @return The sum of a and b.
 */
int add(int a, int b) {
  return a + b;
}
```

There are many comment formats that Doxygen understands, as well as many more tags.
Doxygen is configured via a file called Doxyfile. In new projects, we can generate a default one by running:

Source code

```
$ doxygen -g Doxyfile.in # The .in is important, we will use CMake to generate
↪  the final file
```

This file is mostly ok for us as it is, but we want to change a couple of variables. Open this file and look for the following variables:

- `OUTPUT_DIRECTORY`. Set to `@DOXYGEN_OUTPUT_DIR@`.
- `INPUT`. Set to `@DOXYGEN_INPUT_DIR@`.

Variables between @ are CMake variables that we will replace with the actual values in the CMakeLists.txt file.

3.2.1 Adding it to CMake

Doxygen is a standalone tool that you can run from the command line. But we can also integrate it into CMake to make it part of the build process. This way, every time you build your project, the documentation will be updated.
Create a CMakeLists.txt file in the docs folder with the following content:

Source code

```
find_package(Doxygen REQUIRED)

set(DOXYGEN_INPUT_DIR ${PROJECT_SOURCE_DIR}/src) # Location of the source code
set(DOXYGEN_OUTPUT_DIR ${CMAKE_CURRENT_BINARY_DIR}/doxygen) # Location of the
↪  generated doxygen files
set(DOXYGEN_INDEX_FILE ${DOXYGEN_OUTPUT_DIR}/xml/index.xml) # The main file
↪  that Doxygen generates

set(DOXYFILE_IN ${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile.in) # The template Doxyfile
set(DOXYFILE_OUT ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile) # The output Doxyfile
configure_file(${DOXYFILE_IN} ${DOXYFILE_OUT} @ONLY)
# Doxygen won't create this for us
file(MAKE_DIRECTORY ${DOXYGEN_OUTPUT_DIR})
# Only regenerate Doxygen when some public headers change
get_target_property(PUBLIC_HEADER_DIR my_library INTERFACE_INCLUDE_DIRECTORIES)
file(GLOB_RECURSE PUBLIC_HEADERS ${PUBLIC_HEADER_DIR}/*.h)
add_custom_command(OUTPUT ${DOXYGEN_INDEX_FILE}
                   DEPENDS ${PUBLIC_HEADERS}
                   COMMAND ${DOXYGEN_EXECUTABLE} ${DOXYFILE_OUT}
                   MAIN_DEPENDENCY ${DOXYFILE_OUT} ${DOXYFILE_IN}
                   COMMENT "Generating docs"
                   VERBATIM)
```

This CMakeLists.txt requires a file called Doxyfile.in in the same folder. Files with ".in" are CMake configure files, which means that special variables in the file will be replaced by CMake variables to generate a final file (in this case called Doxyfile).

If you run cmake now (from the build directory) a bunch of things will be generated in the docs folder, including an index.html file that you can open in your browser to see the documentation. In the next section we will make this prettier.

3.2.2 Key new concepts

Info

Doxygen
A tool that can generate documentation from specially formatted comments in the code.

Info

CMake configure_file
A CMake command that copies a file to another location and replaces variables in the file with CMake variables.
configure file will look for variables in the form @VAR@ and replace them with the value of the CMake variable VAR.

3.3 Sphinx and breathe

I recommend you install sphinx and breathe via pip. Simply add them to your environment file for a project.
A minimum Sphinx project consists of a source folder with a conf.py file and a index.rst file. The index.rst file is the entry point to the documentation and the conf.py file contains the configuration for the project. You can create a new Sphinx project by running this in the docs folder:

Source code

```
$ sphinx-quickstart
```

Advice

You will customize conf.py as time passes and you learn. In practice when you start a new project you will copy-paste the conf.py file from a previous project and adapt it to the new one.

This is a minimal conf.py that will work for us:

Source code

```
project = "MyVeryCoolProject"
copyright = "2024, You"
author = "You"

# Breathe is an extension that allows us to connect Doxygen with Sphinx
# There are many sphinx extensions, which can be added to this list
extensions = ["breathe"]

# Breathe Configuration
breathe_default_project = project
templates_path = ["_templates"]
exclude_patterns = []

html_theme = "alabaster" # There are many themes to choose from
html_static_path = ["_static"] # This is where you put static files like images
```

I like to put the sphinx documentation inside a "source" folder within the "docs" folder.
Along conf.py we will have a index.rst file, which contains the actual web page content. Here is a minimal index.rst file:

Source code

```
Welcome to MyVeryCoolProject's documentation!
=============================================

Some welcoming text here, perhaps a introduction to your project.

.. toctree::
   :maxdepth: 2
   :caption: Contents:

   introduction
   modules

Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

reST is a somewhat intuitive format, you will get the hang of it quickly as you need stuff. The primer is a good reference.
The toctree directive is used to create a table of contents. The maxdepth option specifies how many levels of subdirectories to include in the table of contents. The caption option specifies the title of the table of contents. The toctree directive is followed by a list of documents to include in the table of contents. For instance "introduction" expects a file called introduction.rst in the same folder as index.rst. Any headings in the included files will be used as section titles in the table of contents.

3.3.1 Adding it to CMake

We must run sphinx-build after Doxygen has generated the documentation database. We can do this by adding a custom target to the CMakeLists.txt file in the docs folder:

Source code

```
set(SPHINX_SOURCE ${CMAKE_CURRENT_SOURCE_DIR}/source) # Location of the Sphinx
↪  source files (index.rst, conf.py,...)
set(SPHINX_BUILD ${CMAKE_CURRENT_BINARY_DIR}/sphinx) # Location of the
↪  generated Sphinx documentation
add_custom_target(Sphinx ALL
                  COMMAND
                  sphinx-build -b html
                  -Dbreathe_projects.${PROJECT_NAME}=${DOXYGEN_OUTPUT_DIR}/xml
                  ${SPHINX_SOURCE} ${SPHINX_BUILD}
                  WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
                  COMMENT "Generating documentation with Sphinx")
```

With some luck and a bit of tweaking, you should now be able to run `cmake --build .` from the build directory and get a nice web page with your documentation. You will find the web page in the docs/sphinx folder. Open the index.html file in your browser to see the documentation.

Advice

It is now easy to upload this documentation to a service like ReadTheDocs and have it automatically updated every time you push to a github repository. Check out an example brought to you by yours truly:

- TorchMD-Net documentation source and TorchMD-Net documentation

The source code for ReadTheDocs itself follows a similar structure.

3.3.2 Adding our documented functions to the web page

Thanks to breathe you will have some new directives available in your reST files. For instance, you can include a function in the documentation like this:

Source code

```
.. doxygenfunction:: my_library::add
```

You can also include the docs for all the functions in a namespace like this:

Source code

```
.. doxygennamespace:: my_library
```

There are a bunch of these.

3.3.3 Key new concepts

Info

Sphinx
A tool that can generate documentation from reStructuredText files.

Info

Breathe
An extension for Sphinx that allows us to connect Doxygen with Sphinx.

Info

reStructuredText
A markup language that is used to write documentation in Sphinx.