EEEE2076 -Software Development Group Design Project

Worksheet 5 - Automatic Code Documentation and Installer Packaging

P Evans

February 5, 2026

Contents

1 Documentation

You should document your code, documenting code means writing explanations that describe how to use the code - things like describing what a function does, what arguments it takes, its return value. This is important as it allows other developers to build on the code at a later date, it is especially important for code that is designed to be reused by other developers e.g. libraries such as Qt and VTK.

Documentation is usually written by the addition of comments to the code when it is being written - it is easier to add some comments explaining how code works when you are writing the code. These comments are then post-processed to produce documentation in a useful format (e.g. a pdf or a web page). A number of tools are available to do this post-processing and the documentation generator we will be using is ”doxygen”. It is still the most popular solution to document c++ heavy projects and you will have more than likely encountered the output of this system online.

1.1 Exercise 1 - Doxygen

Install Doxygen on your system (just Google Doxygen). Create a Worksheet 5 folder and copy across *adder.cpp adder.h calc.cpp* from Worksheet 5. Add Doxygen comments to *adder.h* according to the following instructions and use Doxygen to produce HTML documentation.**

Here is an example of how to decorate your code with comments that will be considered documentation strings by ”doxygen” - the statements in [] brackets are just explanation of what each line is, and wouldn’t normally be included.:

```
// adder.h
#ifndef MATHSLIB_ADDER_H
#define MATHSLIB_ADDER_H

/** @file
* This file contains the declarations of all exported functions in the maths library.
*/

/** Adding function                        [Brief description]
*  This function adds two numbers             [More detail]
*  @param a is the first number          [Parameter definition]
*  @param b is the second number         [Parameter definition]
*  @return sum of a and b                [Return value description]
*/
int add(int a, int b);

#endif
```

A comment meant for processing by Doxygen sets itself apart from any old comment by the opening sequence `/**`. In addition there are a number of keywords that help Doxygen interpret the role and meaning of the documentation comment. In this example, *@file* indicates that this string contains information on the file itself (as opposed to one of its comprising entities). The second string in this example above is by virtue of its placement (directly above a function declaration) interpreted as a description of that function. As a result, doxygen will place this comment in the appropriate location in its output.

Important: If no `/** @file ... */` block is provided, Doxygen does not pick up on any of the function documentation comments in the file. This is because there is no natural place for that documentation to be included in the output. Documenting the file will generate a corresponding page in the output and all documentation for functions in that file will be included on that page.

The first step in using Doxygen is to generate a configuration file at the root of your project:

```
C:\Users\ezzpe\2025_ezzpe\Worksheet5>doxygen -g
```

A file called *Doxyfile* will appear, you probably don't need to modify this immediately if it was created inside your Worksheet5 folder as by default, doxygen seraches the current folder for source files. If your Doxyfile isn't in the same folder as the source code, open this file and search for the "INPUT" setting - modify this to contain the path to your source code. e.g. if you put the Doxyfile in the root of the repository, you would need to modify the Doxyfile to tell doxygen to look in the Worksheet5 subfolder.:

```
INPUT = Worksheet5
```

With these settings, simply run ‘Doxygen‘ in the folder where the ‘Doxyfile‘ exists.

```
C:\Users\ezzpe\2025_ezzpe\Worksheet5>doxygen
Searching for include files...
Searching for example files...
Searching for images...
Searching for dot files...
Searching for msc files...
Searching for dia files...
Searching for files to exclude
Searching INPUT for files to process...
Searching for files in directory
...
...
Generating hierarchical class index...
Generating graphical class hierarchy...
Generating member index...
Generating file index...
Generating file member index...
Generating example index...
finalizing index lists...
writing tag file...
Running dot...
lookup cache used 1/65536 hits=1 misses=1
finished...
```

A subdirectory named *html* should have been created under your repo's root. This is a good time to add the html subdirectory to your *.gitignore* file. Doing this will avoid the generated documentation to be included in your git commits.

Inspect the *html* subdirectory in file explorer. Double click *index.html* to open that file in your browser. There is not much there (because we only included two comments) but you already get an idea about how your inline comments are processed by doxygen. If this works then add your Doxyfile to Github as it will be needed later.

Once you have verified that Doxygen is working, delete the html folder it created before moving on

1.2 Automatic creation of a documentation website

The aim is now to have Doxygen and Github automatically update and publish the documentation website each time code is updated. First you need to setup Github so that it is able to publish documentation web pages.

1.2.1 Exercise 2 - Setting up your Github.io website

In order to automate the process we will leverage the ability to run any script we want on the virtual machines spun up by a Github feature called Actions, this can include running Doxygen.

The resulting publication will be published on GitHub. This is possible because GitHub creates a publicly visible web page for every repository at:

https://<username>.github.io/<reponame>

The content GitHub puts on that web page is the tree on the specially named branch *gh-pages*. Typically there is little or no overlap between the contents of the code repository and what you'd want to display on the project's home page. Even though this is not what git is designed to do, there is nothing stopping you from putting wildly different content on two separate git branches. To explain to the system that 'gh-pages' shares no content and thus no history with *master* and the other branches we designate this branch as *orphaned*: this means there is no parent commit to the first commit on this branch. In other words it does not share history with another branch. Open a terminal and cd to the root of your individual repository, then:

```
C:\Users\ezzpe\2025_ezzpe>git checkout --orphan gh-pages
C:\Users\ezzpe\2025_ezzpe>git rm -rf .
```

Note the line git rm –rf deletes everything in the repository, this is ok if you have switched to the new gh-pages branch but **Make sure your are actually on the gh-pages branch (git status) before you run this command.** Removing all files on this branch is what you want since it will be populated with the output of Doxygen. It is safe because the source code is securely stowed away in the *master* branch, which is unaffected by whatever happens on this orphaned branch. At this point you'll want to make sure all files have been deleted from your repository. git rm –rf will only have deleted files that were tracked by Github so you'll need to make sure you delete any other files, either by selecting and deleting them in the file browser or using ‘del‘/‘rm‘. Once you are happy the gh-pages branch is empty you'll need to add a README to the branch, add, commit and push.

```
C:\Users\ezzpe\2025_ezzpe>echo "gh-pages branch" > README.md
C:\Users\ezzpe\2025_ezzpe>git add .
C:\Users\ezzpe\2025_ezzpe>git commit -a -m "Clean gh-pages branch"
C:\Users\ezzpe\2025_ezzpe>git push origin gh-pages
```

Add a simple *index.html* file and commit it to *gh-pages* - type some sort of message in it so you can verify that it has uploaded later. Push to the GitHub hosted remote and inspect the result at *https://<username>.github.io/<reponame>*. This site is public, don't upload content unfit for distribution!

1.2.2 Exercise 3 - Automating the publication of documentation

Important: After finishing the previous exercise, switch back to the master branch (git checkout master). The aim of this exercise is to add a script to Github that tells it to run Doxygen whenever you push updates to the code, and send the output of Doxygen to the gh-pages branch. Github has a feature called Github Actions, this is a Continuous Integration (CI) system. CI systems are basically virtual computers that can be triggered to perform set tasks. To make Github Actions work you need to create a ‘.github/workflows‘ folder in the root of your repository and within this folder, you create a ‘.yml‘ file - this file is a script that defines an action and when it should be triggered.

The tools used for documentation generation will be run on the source code of your project, so any related scripts and configuration files need to be stored in the master branch. The output of the documentation generator, on the other hand, will be committed on the gh-pages branch.

The aim now is to have Github automatically publish your documentation to the Github.io website. To do this, create a file called doxygen-deploy.yml with contents as given below. This file then needs to be added to the *.github/workflows* subfolder.

```
# doxygen-deploy.yml
name: doxygen-deploy

on:
  push:
    branches: [ master ]       # Note: this needs to be set to your default branch
                           # name (this might be "main" rather than "master"



jobs:
  # The job that will build worksheet 5
  deploy-worksheet5:

    # Defines the operating system for the virtual computer
    runs-on: ubuntu-latest

    # A list of steps to follow to complete the build
    # (a list of commands to execute on the virtual computer)
    steps:

    # This is a predefined action that checks out a copy of your code
    - name: Checkout
```

```
      uses: actions/checkout@v2

    - name: Install doxygen
      run: sudo apt-get install -y doxygen

    # Run Doxygen on Worksheet5 folder
    - name: Run Doxygen
      working-directory: ${{github.workspace}}/Worksheet5
      run: doxygen

    # Deploy html output from Doxygen to ghpages branches- name: Deploy
    - name: Deploy
      uses: JamesIves/github-pages-deploy-action@v4.2.5
      with:
        # The branch the action should deploy to.
        branch: gh-pages
        # The folder the action should deploy.
        folder: ${{github.workspace}}/Worksheet5/html
```

Important: the *.yml* format uses indentation as an integral part of the script (indentation performs the same job as {} brackets in C/C++). Each indent level in the file above is 2 spaces wide - you should make sure your file is the same, if your indentation is not 2 spaces, not consistent throughout the file, or if you use tabs for indentation, you may get errors.

Add, commit, push and you should be able to view the documentation on Github.io. If it doesn't appear at first: - Check the *Pages* settings under the repository *Settings* tab. Make sure the Github pages feature is enabled and that the gh-pages branch is set as the source. Github pages won't if you didn't register for the Github educational pack as you were supposed to at the start of the lab sessions. - Keep trying, it can take a minute or so for Github.io to update.

You may get an error related to respository permissions. In this case, go to the repository settings, then Actions, then ensure Actions have read and write permissions.

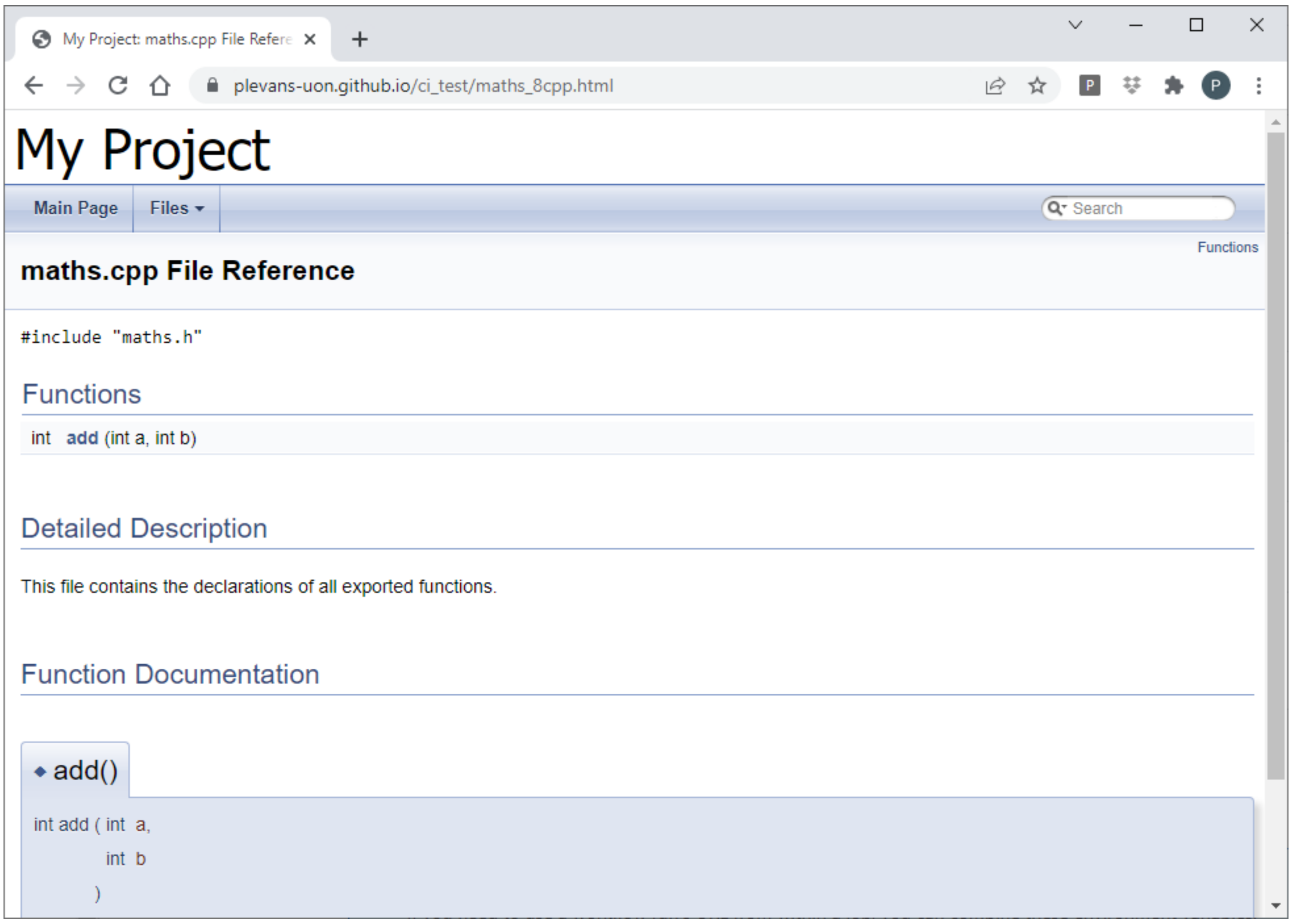
For your group work - you are aiming to have a full set of code documentation on Github.io which will involve adding the relevant comments to the source files but you may also need to edit the Doxyfile to customise Doxygen output. You could also add a action status indicator for documentation generation.

2 Creating an Installer

Most Windows applications will come bundled in an installer that will copy relevant files, create the necessary directories and perform other functions such as adding items to the Windows start menu or creating environment variables.

For now we have left the outputs of the build process inside the build directory (which we typically created as a subdirectory of the source directory). During development this is a perfectly fine mode of operation. As you arrive at the stage where you'd wish to deploy your software to a client, a different approach is required for the following reasons:

- You do not want to deploy the entire build directory which contains, in addition to the executables and libraries, also the intermediary build files and CMake configuration files.
- You want to let the user make decisions about what components to install. This way, basic users can


My Project: maths.cpp File Reference

plevans-uon.github.io/ci_test/maths_8cpp.html

My Project

Main Page | Files | Search

Functions

maths.cpp File Reference

```
#include "maths.h"
```

Functions

int **add** (int a, int b)

Detailed Description

This file contains the declarations of all exported functions.

Function Documentation

◆ add()

```
int add ( int  a,
          int  b
        )
```

Figure 1: Automatically created documentation website

save disk space whereas advanced users and client programmers can choose to install also header files and import libraries.

- The layout of the build directory depends on the build system. To keep your package as portable as possible you'd want to adhere to a more conventional layout that is (almost) the same, regardless the target platform.

To do this you will use *cpack*, cpack is yet another component of the cmake ecosystem. It provides a convenient way to gather all the executables, libraries, etc. of your package and copies them in the default directory structure for the desired platform. Next it store this directory structure in an archive together with instructions on how to unpack it at the client site. The delivery medium can be as simple as a zip-file, but more user friendly options such as the very familiar Windows style NSIS installer are also an option.

Fortunately, installation and packaging is a very simple using the features cmake provides. Consider the following *CMakeLists.txt*, which is a modified version of the file you should already have in your Worksheet5 directory:

```
if(WIN32)
    set(CPACK_GENERATOR "NSIS")
```

```
else()
   set(CPACK_GENERATOR "ZIP")
endif()

include(CPack)

add_library(maths adder.cpp)

add_executable(calc calc.cpp)
target_link_libraries(calc maths)

install( TARGETS calc maths
         RUNTIME DESTINATION bin
         LIBRARY DESTINATION lib
         ARCHIVE DESTINATION lib/static)
```

Most of the above script you have already encountered. There are statements that define targets and inform CMake what sources and libraries these targets depend on. What is new here is the lstinline[language=Perl]install command, and the lines up to include(CPack). The install command allows you to list which targets are part of the install package and what the default subdirectories for the various types of outputs are. The above tells CMake to put any runtime components (.exe and .dll on Windows) in a subdirectory called bin and library components in a subdirectory called lib. This structure is common on Unix derived systems (e.g. Linux, MacOS, Android) where it forms the basis of the file system, but many Windows applications that use Open Source libraries will recreate a Unix-like subdirectory structure in the install directory.

On Windows, an extra program is required to create the familiar Windows style installer. This program is called the NullSoft Installer System (NSIS) - search for this, download and install now.
The above modification will cause CMake to create additional build targets in the Visual Studio project called INSTALL and PACKAGE. If you right click on PACKAGE in VS and select *build* it will build a stand-alone installer, if you right click on INSTALL it will run the install process.

2.0.1 Exercise 4 - Modify your CMakeLists as indicated above, rerun CMake and Visual Studio and check you can create and run the installer from within in Visual Studio.

Important Note: Installing a program on Windows now requires "Administrator" privileges. If you try to run the install target from within Visual Studio, the install process will inherit Visual Studios privileges which are very unlikely to be at admin level so the install fails. You can resolve this by either building the package, then running this from outside VS, or by reopening VS with admin privileges (right click in its icon, run as administrator).

Upload your installer as a Github release

2.1 Final Points

During CMake installation and packaging, only components of the current project are considered. In your final product you will most likely also want to ship the Qt and VTK dlls your executable has runtime dependencies on. CMake provides some support in automatically including these during installation/packaging. Look into the help of CMake modules InstallRequiredSystemLibraries and DeployQt5.cmake. You can also use the dependencies tool to trace which library components are required by your application Don't expect to get this right the first time around. This is one of the more difficult tasks in the group exercise.