# CS 1XC3 Lab 9: Documentation using Doxygen

Mar. 27th-31st, 2023

## Contents

## Introduction

In this lab exercise, we will explore the document generation capabilities of Doxygen, as well as how to automate documentation generation, and how to host documentation using Github Pages.

### Setup

In order to set up your "CS1XC3" repository for this laboratory exercise:

1. Create a new directory, "L09".
2. Download the compressed archive "L09.tar.gz" from the Avenue content.
3. Decompress the archive inside the L09 directory.
4. Put your answers in the file templates in the decompressed directories.

You will also need to have Doxygen installed on your system (`https://www.doxygen.nl/download.html`). If you're using Pascal, it has already been installed for you.

## 1 Warmup Activity: Markdown Tutorial [2 points]

Doxygen uses markdown-style document formatting. Markdown formatting is an increasingly popular and minimalistic way of formatting plain text documents. Best of all, a document that is Markdown formatted is highly readable both rendered and in plain text. Please complete the following tutorial on Markdown formatting:

- `https://www.markdowntutorial.com`

Create a file "part1.txt" in "CS1XC3/Lab09", and answer the following question:

- In the list of links, the fourth link has a cheat sheet. What is the 8th markdown element in the first table?

## 2 Warmup Activity: LaTeX-style formulas [5 points]

Both Markdown and Doxygen support LaTeX-style formulas. In this exercise, you will create LaTeXformatted formulas for the following equations. Save these formulas to a file "part2.tex", and commit it to your CS1XC3/L09 repo when you're finished.

In LaTeX, inline formulas are enclosed by $ single dollar signs $, and centered formulas are enclosed by $$ double dollar signs $$. Please use either. It is recommended that you develop these formulas in the following interactive Latex editor:

- `https://arachnoid.com/latex/`

| | |
|---|---|
| Freedom is the freedom to say that | $2 + 2 = 4$ |
| Quadratic Equation | $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ |
| Integration by parts | $\int u dv = uv - \int v du$ |
| Fourier Transform | $\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx$ |
| Modus Ponens | $P \rightarrow Q, P \vdash Q$ |

Hints:

- You are not required to understand these formulas, just to transcribe them.
- To write symbols in LaTeX, you use the backslash character to start the symbol, and then a series of characters which indicate the symbol to be entered. For example, to write the greek letter $\pi$, you enter "\pi".
- A table of the names of the symbols used in the above equations is provided below (excluding the obvious ones), but you will need to look up their usage yourself using your favourite internet search engine!

| Symbol | Name | Category |
|---|---|---|
| $\pm$ | plus/minus | Binary Operator |
| $\int$ | Indefinite Integral | Calculus Operator |
| $\int_b^a$ | Definite Integral | Calculus Operator |
| $\xi$ | Lower-case Xi | Greek Letter |
| $\pi$ | Lower-case Pi | Greek Letter |
| $\hat{x}$ | Circumflex | Math-mode Accent |
| $\rightarrow$ | Implication | Logical Operator |
| $\vdash$ | Turnstile ("Proves") | Logical Operator |

## 3 Activity: Setting Up Our Makefile [5 points]

Let's configure our project to build automatically using the power of `make`!

1. Create the file "Makefile" in the root directory of your "CS1XC3/L09" repository.
2. Using your favourite text editor, add rules to compile the files "main", student.o" and "course.o" from the source code files "main.c", "student.c" and "course.c", in the same manner as Lab 7.
   - Have the makefile put all object and executable files in a "build/" directory, and have the makefile create that directory if it doesn't exist. If you use any shell scripts, create a "scripts/" folder in the repository for them and keep them there, so as not to clutter our root directory.
3. Have your makefile create a "documentation" directory as well, and a "docs" subdirectory within that directory, if niether exist. Make the existance of these directories a prerequisite of "build/main".
4. Add a `clean` phony target which removes the build directory completely if it exists.

5. If it isn't already, make sure that "build/main" is the default target.

6. OK! Now let's get automatic document generation working! Recall that Doxygen needs a configuration file called a "Doxyfile" to work, and that we generate said file using:

   ```
   $ doxygen -g
   ```

   We can use our Makefile to build this file if it doesn't exist! Write a new rule in our Makefile. The target will be the doxygen configuration file, and we can use the above command as our recipe.

7. We generate our documentation using:

   ```
   $ doxygen doxyfile
   ```

   Later on, it will be useful to be able to generate our documentation *without* also compiling our program, so it makes sense to make this a separate rule, rather than lumping it in with our compilation command. Add a phony target to the makefile called "docs", which has as a prerequisite the existance of the doxyfile, and uses the above command to generate the documentation. Add this rule as a prerequisite to "build/main".

8. Run the make file in such a way as to only generate the doxyfile.

9. Open up the generated Doxyfile so we can get configuratin'!

   - You'll notice that the Doxyfile is very long ($\sim$ 2500 lines!), but you'll also notice that, true to form, it is very well documented. We will need to change some of the options in this file to configure it for our project. It would be worth learning at this point how to search the doxyfile using whatever text editor you're using (the emacs shortcut is Ctrl+s, which you'd think would be save, but save is Ctrl+x, *then* Ctrl+s. That's emacs for you!).

     The student is encouraged to read the descriptions of these options as they modify them.

     | Option | Value |
     |---|---|
     | `PROJECT_NAME` | `"CS1XC3 Lab 9"` |
     | `PROJECT_BRIEF` | <provide a brief description of what you think the point of this lab is.> |
     | `OUTPUT_DIRECTORY` | `documentation` |
     | `INPUT` | `src` |
     | `RECURSIVE` | `YES` |
     | `GENERATE_HTML` | `YES` |
     | `HTML_OUTPUT` | `docs` |
     | `GENERATE_LATEX` | `NO` |
     | `HAVE_DOT` | `YES` |

     Some of these options will already be set to the above values, but please read the documentation about them anyways. The options specifying which directories to use are of particular importance.

10. Run the whole system to make sure everything is working.

11. Commit your repo to finish this part.

If you are working on a linux system with a GUI, you could at this point open up your generated documentation (i.e., open "index.html" with a web browser), and see what Doxygen is able to pick up without any doxycomments at all! If you aren't, I recommend cloning your repo somewhere that does (even Windows would work for this).

## 4 Activity: Setting Up Github Pages [3 points]

First, let's set up a Github action, so that our documentation will automatically be updated whenever someone pushes something to the repo.

1. In the actions tab from your github page for your "CS1XC3_L09" repo, select **C/C++ with Make** by clicking "set up this workflow".

2. Now we need to modify our workflow file to use our make file. You'll find if you just remove all the steps but one, and modify it to invoke `make docs`, Github will complain because Doxygen isn't installed!

3. Fortunately, we can install doxygen the same way that we would in an ordinary Ubuntu system.

   ```
   sudo apt-get install doxygen
   ```

   Github actions operate inside a virtual machine instance which is spawned specifically for the action, and which is destroyed afterwards. These virtual machines pose no security risk to Github because they're sandboxed, so Github very graciously gives us sudo priviledges for them. All we have to do therefore, is to run the command installing doxygen before we invoke `make`. HINT: `https://stackoverflow.com/questions/59954185/github-action-split-long-command-into-multiple-lines`

4. Commit the YAML file, and check that the action executes with no errors.

5. Good work!

Unfortunately, that's as far as we can go without making our repositories public (which of course would be in violation of our academic integrity policy). In the future, if you're working on a project that isn't required to be a private repository, you can automatically configure Github to use your generated HTML pages, and you can even set up a custom URL for it (within certain limitations).

- `https://docs.github.com/en/pages/getting-started-with-github-pages/configuring-a-publishing-source-for-your-github-pages-site`

It's super easy; all you have to do is specify which branch and directory the HTML files are in.

## 5 Documenting Structs [6 points]

Alright! Now that we're finished with all the setup, let's actually document some code!

### 5.1 `structs` in C

If you've taken a look at the documentation doxygen provides automatically, you'll notice that under the "Classes" tab, we have two structures, `_course` and `_student`.

- The word "class" here is a misnomer. C doesn't have classes, but this documentation was built for C++, so it misinterprets structs as a type of class.
- The documentation also lists the fields of these structures as being "public attributes", when in fact that distinction doesn't exist in C.

A struct, briefly stated, is a collection of variables which have all been packaged up as one data type. The members of a struct are accessed via dot notation, which is the same way attributes are accessed in Python. Structs could be thought of as classes without the methods.

### 5.2 Documentation

Reading the generated pages, we can see that `_course` and `_student` are being generated from "src/course.h" and "src/student.h" respectively.

1. Remember the form for a doxycomment:

   ```
   /**
    * \brief A brief description
    *
    * This is a Doxycomment, explaining everything you need to
    * know about the thing immediately after this block.
    *
    */
   ```

   Place a doxycomment at the beginning of the struct block. Within it, provide a description of the structure. What sort of data would you suppose it to hold, based on its name, and the name of its members? Provide a brief description which summarizes the above.

2. The following format is used to place a doxycomment on the same line as the object it documents:

   ```
     int somevar; /**< A description of somevar in plain english */
   ```

This style of documentation is commonly used within structures, since documentation of each attribute tends to be short, and this style is less disruptive of the readability of the file as C code.

Write short descriptions of each member using the above style of doxycomment. When you're finished, document the structure in "course.h" in the same manner.

- NOTE: Although we have set up github to automatically run doxygen, those changes won't appear in a local clone, even if we try to pull them. To build your documentation locally, you still need to use `make docs` from the command line.

Commit your work to finish this part.

## 6 Documenting Functions [18 points]

When it comes to documenting our functions which aren't in "main.c", there are two places we could put our doxycomments:

1. The source code file
2. The header file

While it might seem intuitive to document our source code adjacent to the actual implementation, we must remember what the purpose of a header file is. Header files provide the interface to our source code file. When a developer is in doubt about how to use one of our functions, the first place they should check is the header file (in the absence of proper documentation of course). Therefore, it is the header file, not the source code file, in which we should put our doxycomments.

### 6.1 Tags Galore!

Just like the brief tag used above, we can tag certain parts of our function documentation, so that doxygen can more intelligently assemble the resulting documentation. You will be expected to use the following tags (wherever appropriate) in your documentation.

| Tag | Description |
|---|---|
| `@param <name> <description>` | Documents a given parameter (or argument) to the function. |
| `@return <description>` | Documents the return value of the function. |
| `@see <element>` | Creates a symbolic link to another code element that may be of interest. |
| `@note <description>` | Information that may be of interest. Gets a callout box. |
| `@attention <description>` | Information that is of interest. Gets a callout box. |
| `@warning <description>` | Information vital to correct operation. Gets a callout box. |

For example, if I were to document a function from a previous lab...

```
/** \brief Displays Barrett's Privateers by Stan Rogers
 * Ideally set to music, this function is a real-time recitation of
 * the classic Canadian sea shanty "Barrett's Privateers" by
 * Stan Rogers.
 * @param pirates The number of pirates to embark.
 * @return The weight of American gold the pirates plundered (always zero).
 * @see northwestPassage, theWreckOfTheEdmundFitzgerald
 * @note The pirates should be as inexperienced as possible when passing to the function
 * @attention The sea is like a cruel mistress. You can love her, you can hate her, but
   you can never trust her.
 * @warning Americans are a pretty ornery bunch, and don't give up their gold easily.
 */
float barrettsPrivateers (int pirates);
```

Both "course.c" and "student.c" contain four functions each. Document each of these functions their respective header files. When you're finished, document the main function in "main.c".

### 6.2 File Headers [6 points]

In addition to the tags used for function declarations, there are some special tags specifically for file headers. Now that we've documented all of the structures and functions in our program, we should have a vague idea what the files are for.

| Tag | Description |
|---|---|
| `@file <filename>` | |
| `@author <Your Name>` | |
| `@date <date created>` | |
| `@brief <description>` | |

To each file (*.c and *.h) add a file header which uses the above tags. Provide a description of the purpose of the code in each file (it's OK if a *.c file's description overlaps somewhat with its corresponding *.h file.)

Buid your documentation and check it to make sure everything ended up in the right place.

## 7 README Integration [10 points]

It has been conventional since at least the early 1980s for software packages to include a "README" file. The README, as the name would imply, contains important information about how the software is to be installed and run, as well as anything else of interest to the user. In some sense, the README can be considered the user manual for the software. A README file should therefore be written for the **end user**. README files are typically left in the top-level directory of the software project.
While it is still quite common for a README file to be unformatted plaintext, markdown is becoming popular for READMEs, since even unrendered markdown is highly readable in a plaintext environment. Both Github and Doxygen work with README files relatively easily.

- In Github, any file named README (with a variety of file extensions) will automatically be loaded as the front page for the repository. The front page information appears directly under the file heirarchy view in the code tab. Convenient!
- In our Doxyfile, if we set `USE_MDFILE_AS_MAINPAGE` to "README.md", and include the line:

  ```
  INPUT += README.md
  ```

  Doxygen will be configured to use our README file as it's main page, which up to now has been blank.

Create a README file for this software package. The readme should minimally discuss the following things:

- date, author, project name
- description of the purpose / function of the software
- installation instructions
- configuration instructions
- operating instructions
- troubleshooting
- changelog

It's OK to make some of the above up for this lab (for example, you can't know anything about this software's changes over time, since you didn't write it.) However, you are going to be graded on your presentation, and how well your README renders in Doxygen and Github, so make it look nice! Use the formatting we learned in Activity 1 in this lab.