Documentation

Ramses van Zon

PHY1610H 2026 Winter

Code is for humans

Programs are meant to be read by humans and only incidentally for computers to execute.

(Harold Abelson, *Structure and Interpretation of Computer Programs*)

How so?

- Programmers spend more time reading someone else's code than writing their own.
- There's no such this as a one-off script.
  (If you have saved a script as a file, it's no longer one-off, and you or someone else will eventually use it again and want to adapt it.)

Readability, documentation, and maintainability are very important.

How to code for humans:

1. Write clear code
2. Comment your code
3. Document your code

How to code for humans 1: Write clear code.

KISS

"Keep It Simple, Stupid!"

- Do not write code that is more complicated than necessary.
- It will take too long for the next programmer of your future self to decode.

Debugging is twice as hard as writing the code in the first place.

Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

(Brian Kernighan)

DRY

"Don't Repeat Yourself"

- Use functions and modules to extract repeated code.
- Less code to figure out, less possible bugs, less code to maintain.

Separation of concerns

- Each function should do only one thing, and do it well.
- Each module should have one responsibility.
- Interaction between modules and functions only through function arguments.

This makes the code easier to figure out, bugs become less complex, documentation becomes easier to write, and code can be reused in more situations.

How to code for humans 2: Comment your code

Commenting

Even if code seems simple and to the point:

```
double d(double b)
{
    return 0.5*9.81*b*b;
}
```

But what does it do and why?

Add some comments:

```
// function to compute the vertical displacement
// of an object as a function of time under the
// influence of Earth's gravity
double d(double b)
{
  double a=9.81; //gravitational accelleration
  double c=0.5*a*b*b;//multiply gravity and time
  return c;
}
```

But the last one only describes how, not why.

Commenting the why

```
// function to compute the vertical displacement
// of an object as a function of time under the
// influence of Earth's gravity
double d(double b)
{
  double a=9.81; //gravitational accelleration
  double c=0.5*a*b*b;//multiply gravity and time
  return c;
}
```

```
// function to compute the vertical displacement
// of an object as a function of time under the
// influence of Earth's gravity
double d(double b)
{
  double a=9.81; //gravitational accelleration
  // compute displacement with Newton's equation
  // z = ½at²
  double c=0.5*a*b*b;
  return c;
}
```

Self-documenting code

There's a school of thought that says that if you write clean code, you do not need comments.
We had:

```
// function to compute the vertical displacement
// of an object as a function of time under the
// influence of Earth's gravity
double d(double b)
{
  double a=9.81; //gravitational accelleration
  // compute displacement with Newton's equation
  // z = ½at²
  double c=0.5*a*b*b;
  return c;
}
```

Let's try it:

Better naming: yes!

But this is really verbose, changes the API, and ***still misses the why***.

```
double vertical_displacement_under_gravity(double time)
{
  double earth_gravity = 9.81;
  double displacement = 0.5*earth_gravity*time*time;
  return displacement;
}
```

Commenting is a balance

- Well-named functions and variables to express meaning, but not too long
- Comments that explain the "why".
- Ensure the interface is well defined.
- Comments that explain the how when the above does not suffice

```
// function to compute the vertical displacement of an object
// as a function of time under the influence of Earth's gravity.
//
// parameter(s):
//    time:   the elapsed time in seconds (double)
//
// returns: the vertical displacement (double)
double displacement(double time)
{
  double g = 9.81; // Earth's gravitational accelleration
  // Using Newton's law to compute displacement
  double d = 0.5*g*time*time;
  return d;
}
```

How to code for humans 2: Document your code

Document your modules

The most unlikely pieces of code can end up being reused, so try and add at least a bit of documentation.

There are many documentation styles and philosophies:

- No documentation

  This style also often advocates no comments. The pretense is that clear code is 'self-documenting'.

  Sure, but ... yeah, sorry, no.

- Auto-generated documentation

  Adding specially formatted comments that a tool like doxygen uses to generate documentation.

  This is pretty decent, and a good way to keep documentation up-to-date when the code changes.

- User oriented

  Your code will get read by someone with (much) less understanding of what it's supposed to do than you. If you were in this situation, what documentation would you need to be able to use the module?

  If you do nothing else, at least add a README.md file.
  You can also use Sphinx on top of doxygen to write documentation.

It all starts with comments.

- Yes, code should be as clear and to-the-point as possible.
- But it cannot express why something is done.
- At some point, your code will get read by someone with less understanding than you.
  And this includes your future self.
- Comments will help this person to quickly recall what the code supposes to do, or get familiar with someone's else code.
- One of the things to understand is what each function does, the type and values of the arguments it take, what it returns. as well as limiting cases and restrictions.
- By adding specially formatted comments, one can use tools like doxygen to auto-generate documentation. Doxygen can read your C++ code and combine the definitions of your functions and scripts with your comments, and generate a manual for your code.
- This is a good way to keep documentation up-to-date when the code changes.

Doxygen by Example #1

Doxygen will take your comments and generate documents,
provided they are formatted a certain way.

```
///
/// @brief function to compute the vertical displacement of an object
/// as a function of time under the influence of Earth's gravity.
///
/// @param time the elapsed time in seconds (double)
/// @ returns the vertical displacement (double)
///
double displacement(double time)
{
  double g = 9.81; ///< Earth's gravitational accelleration
  // Using Newton's law to compute displacement
  double d = 0.5*g*time*time;
  return d;
}
```

The /// indicates Doxygen should read these.

The @... marks the meaning of specific parts, like its brief description, parameters, and the return type.

There are other styles that Doxygen allows.

Doxygen by Example #2

```
/// @file   outputarray.h
/// @author Ramses van Zon
/// @date   February 6, 2025
/// @brief  Module for writing a 1d array of doubles to text and binary files.
#ifndef OUTPUTARRAYH
#define OUTPUTARRAYH
#include <string>

/// @brief Function to write an array of doubles to a binary file.
/// This function does a raw dump of the array file to file.
/// @param  s  the filename
/// @param  n  number of elements of the array to write to file
/// @param  x  pointer to the first element of the array of doubles
void writeBinary(const std::string& s, int n, const double x[]);

/// @brief Function to write an array of doubles to a text file.
/// The file will contain each element of the array on a separate line.
/// @param  s  the filename
/// @param  n  number of elements of the array to write to file
/// @param  x  pointer to the first element of the array of doubles
void writeText(const std::string& s, int n, const double x[]);
#endif
```

Doxygen by Example (continued)

1. Generate a configuration file for doxygen called Doxygen (then edit it):

```
$ doxygen -g
$ sed -i 's/PROJECT_NAME[ ]*=.*/PROJECT_NAME=Outputarray/' Doxyfile
```

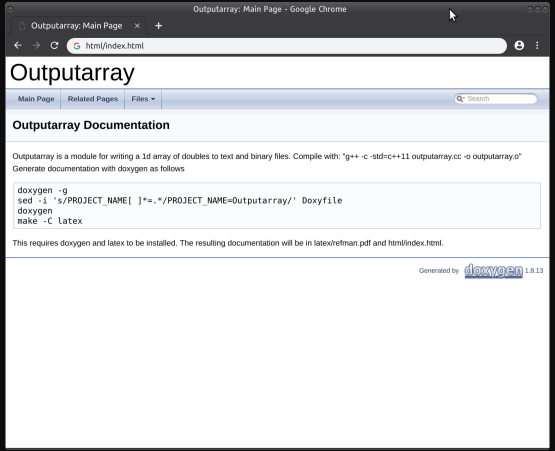
(The sed command just fills in the project name for us into the Doxygen file.)

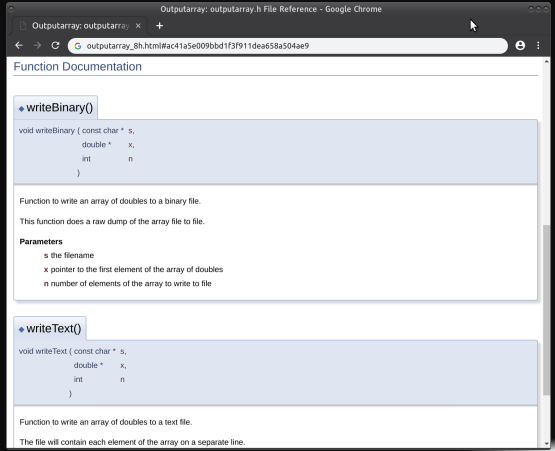
2. Create a README.md

```
[//]: # \mainpage
Outputarray is a module for writing a 1d array of doubles to text and binary files.
Compile with: "g++ -c -std=c++17 outputarray.cc -o outputarray.o"
Generate documentation with doxygen as follows
    doxygen -g
    sed -i 's/PROJECT_NAME[ ]*=.*/PROJECT_NAME=Outputarray/' Doxyfile
    doxygen
    make -C latex
This requires doxygen and latex to be installed.
The resulting documentation will be in latex/refman.pdf and html/index.html.
```

3. Do what the README.md says to generate the documentation in html and latex form

Doxygen by Example - HTML Result





In a Makefile

The command above use sed so that we can automate the documentation creation in our Makefile:

```
doc: outputarray.h
	doxygen -g
	sed -i 's/PROJECT_NAME[ ]*=.*/PROJECT_NAME=Outputarray/' Doxyfile
	doxygen
	make -C latex

.PHONY: doc
```