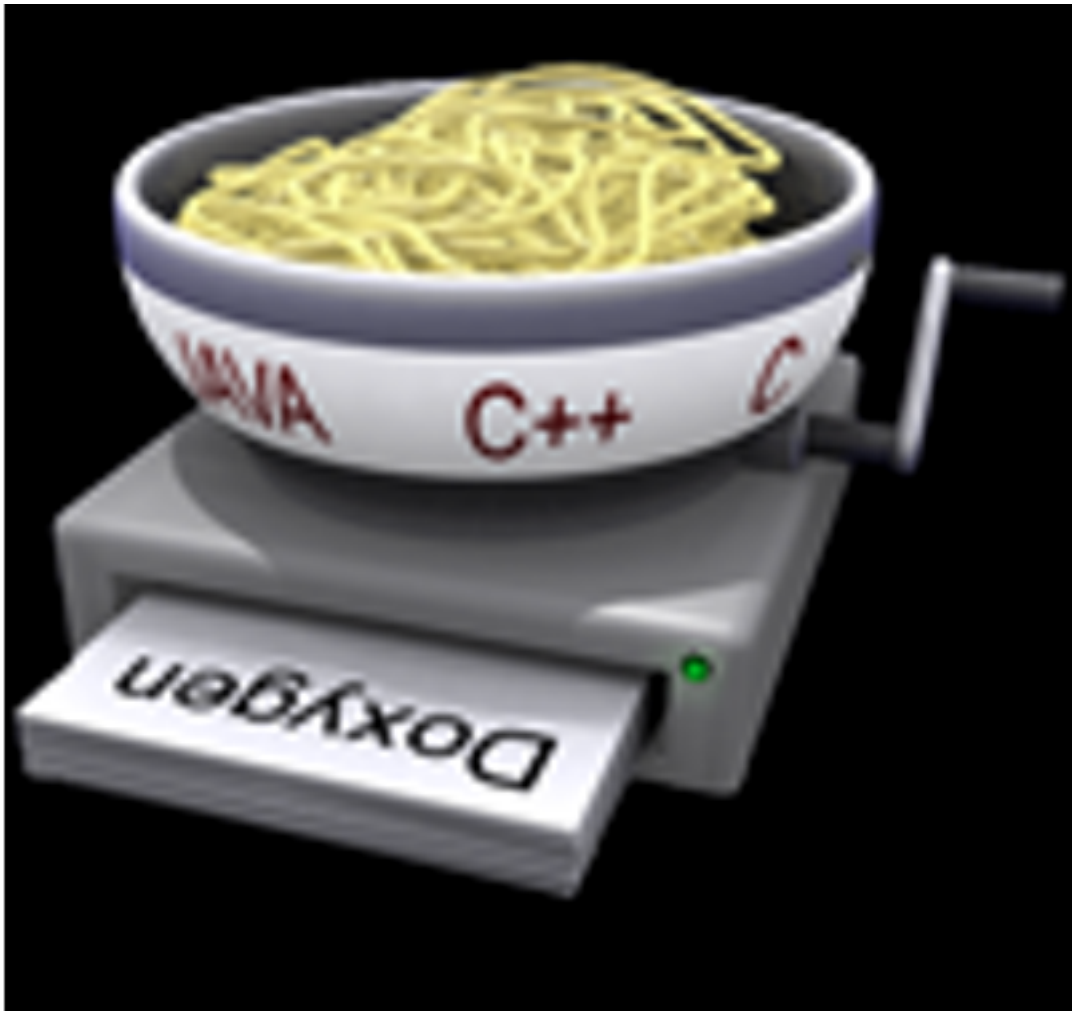
Kyoshe Winstone
MASTER 1 CSMI

INTERNSHIP REPORT



WEEKLY REPORT

0.1 CONTENTS.

This report will focus on the following fields.

i. Introduction

ii. Github

iii. Markdown

iv. Dummy mini-project

v. Doxygen

vi. Refferences

0.2 INTRODUCTION.

Doxygen is a documentation generator, a tool for writing software reference documentation. The documentation is written within code, and is thus relatively easy to keep up to date. Doxygen can cross reference documentation and code, so that the reader of a document can easily refer to the actual code.

This report provides a short quick start on how to produce a documentation of a simple dummy project using doxygen.. Doxygen is used to parse the source code for suitable comment blocks to automatically generate clean Freespace to Open documentation in a number of formats. Doxygen uses a small text file, called a Doxyfile, as a script to analyse source code and produce the required output formats. Doxygen's ease of regenerating documentation from source code allows the production of up to date documentation reflecting the latest source code changes at any time.
Morever, this report introduces basic knowledge on how to make Makefiles and CMakeLists.txt for easy compilation of source codes.
I have also intoduced in this report, the idea of github and how it works.

0.3 GITHUB

GitHub is a web-based Git repository hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features. Unlike Git, which is strictly a command-line tool, GitHub provides a web-based graphical interface and desktop as well as mobile integration. It also provides access control and several collaboration features such as wikis, task management, and bug tracking and feature requests for every project.
GitHub offers both plans for private repositories and free accounts, which are usually used to host open-source software projects. As of 2015, GitHub reports having over 9 million users and over 21.1 million repositories, making it the largest code hoster in the world.

This part of the report covers various areas such as :

I. installing and configuring GIT to allow the github platform to function.

II. Creating an account, signing up and eventually creating Repositories

III. adding files to a repo

IV. Mastering issues

V. pushing commits

VI. pull requests

0.4 MARKDOWN.

Markdown is a markup language with plain text formatting syntax designed so that it can be converted to HTML and many other formats using a tool by the same name. Markdown is often used to format readme files, for writing messages in online discussion forums, and to create rich text using a plain text editor. John Gruber, with substantial contributions from Aaron Swartz, created the Markdown language in 2004 with the goal of enabling people "to write using an easy-to-read, easy-to-write plain text format, and optionally convert it to structurally valid XHTML (or HTML)
Markdown is a lightweight and easy-to-use syntax for styling all forms of writing on the GitHub platform.
During the first part of my internship, i have been able to learn and use the basic steps of using MARKDOWN editor to :

— How the Markdown format makes styled collaborative editing easy

— How Markdown differs from traditional formatting approaches

— How to use Markdown to format text

— How to leverage GitHub's automatic Markdown rendering

— How to apply GitHub's unique Markdown extensions

— Gists on githhub

— Comments in Issues and Pull Requests

— Create Files with the .md or .markdown extension

— Highlighting syntax

```
cmake_minimum_required(VERSION 2.8.9 FATAL_ERROR)
project(operation)
add_library(operationlib operation.cpp operation.h)
target_include_directories (operationlib PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
add_executable(operation main.cpp)
target_link_libraries(operation LINK_PUBLIC operationlib)
```

0.5 DUMMY MINI-PROJECT

The goal of this mini project is to create a simple program and then be able to write the " Makefiles " and "CMakeLists.txt" from the resulting source codes.

— source code for the simple operation program created

operation.cpp

```
/#include "operation.h"
#include <iostream>
using namespace std;
                  //! Default Constructor
                   /*!
                     *constructs an empty object
                    */


operation::operation() : x(0), y(0)
{}
  /**
   *Constructs an objects with two arguments x and y
   */
operation::operation(double x, double y) : x(x), y(y)
{}

  /**
   *set the value of x
   */
void operation::setX(double x)
{
  this->x = x;
}

  /**
   *set the value of y
   */
void operation::setY(double y)
{
  this->y = y;
}

  /**
   *return the value of x
   */
double operation::getX() const
{
  return this->x;
```

```
}

  /**
   *set the value of y
   */
double operation::getY() const
{
  return this->y;
}


  /**
   *returns the sum of two numbers
   */
double operation::addition( )
{
  return (x + y);
}

  /**
   *returns the diff of two numbers
   */
double operation::substruction( )
{
  return (x - y);
}

  /**
   *returns the product of two numbers
   */
double operation::multiplication( )
{
  return (x * y);
}

  /**
   *returns the quotient of two numbers
   */
double operation::division( )
{if(y != 0)
    {
      return x / y;

    }
  //cout << "Error: division by zero.\n";
  //return 0;
}
```

operation.h

```
#ifndef OPERATION_H
#define OPERATION_H

#include <iostream>
using namespace std;
//!operation class
class operation{
  //! Default Constructor
   /*!
    *constructs an empty object
   */
  operation();

  /**
   *Constructs an objects with two arguments x and y
   */
  operation(double x, double y);

  /**
   *set the value of x
   */
  void setX(double x);
  /**
   *set the value of y
   */
  void setY(double y);
  /**
   *return the value of x
   */
  double getX() const;
  /**
   *return the value of y
   */
  double getY() const;


  /**
   *returns a double, The sum of two numbers
   */
  double addition();
  /**
   *returns a double, The difference of two numbers
   */
  double substruction();
  /**
   *returns a double, The product of two numbers
   */
  double multiplication();
  /**
   *returns a double, The quotient of two numbers
   */
  double division();

   /**
    *@param x is an attribute
    *@param y is an attribute too
    *
    */
 private:
        double x,y; /*!<private double values */
```

```
};

#endif
```

main.cpp

```
#include <iostream>
#include <stdlib.h>

using namespace std;

#include "operation.h"

  /**
   *Main class
   *@param argc An integer argument count of the
       command line arguments
   *@param argv An argument vector of a command line arguments
   *Return an integer 0 upon exit success
   */
int  main(int argc, char **argv)
{

  /**
    *@param x is variable
    *@param y is  variable too
    *
    */
  double x =0, y=0;

  if(argc == 3){
    x = atof(argv[1]);
    y = atof(argv[2]);
  }
  /**
   *Creates an object of type "operation"
   */
  operation B(x,y);

    //! Standard output
    /*!
     *Displays the outputs
    */
  cout << B.getX() << "+" << B.getY() << "=" << B.addition() << endl;
  cout << B.getX() << "-" << B.getY() << "=" << B.substruction() << endl;
  cout << B.getX() << "*" << B.getY() << "=" << B.multiplication() << endl;
  cout << B.getX() << "/" << B.getY() << "=" << B.division() << endl;
  return 0;
}
```

— Makefile for our source code

Compiling your source code files can be tedious, specially when you want to include several source files and have to type the compiling command everytime you want to do it. Makefiles are special format files that together with the make utility will help you to automatically build and manage your projects. Below is an example of the makefile for compiling our source codes for the dummy mini-project.

```
all: operation
operation: main.o operation.o
        g++ main.o operation.o -o operation

main.o: main.cpp
        g++ -c main.cpp

operation.o: operation.cpp
        g++ -c operation.cpp


clean:
        rm *o operation
```

How to use the makefile to compile the codes

We recommend creating a new directory and placing all the files in there.

The make utility

If you run

```
make
```

this program will look for a file named makefile in your directory, and then execute it.

If you have several makefiles, then you can execute them with the command :

```
make -f MyMakefile
```

— CMakeLists.txt for our source code

CMake is a sophisticated, cross-platform, open-source build system developed by Kitware in 2001. CMake is the name used to refer to the entire family of tools : CMake, CTest, CDash, and CPack.

CMake ? An intelligent build system used to build simple, to very elaborate projects.

CMake simplifies the potentially monstrous build system into a few easy to write files. It is targeted towards C and C++ and is usable with various compiler/OS support. At the risk of over simplifying things, CMake looks at your project and sees a ?file system ?. You tell it where all your files are at, what you want it to do with them, and it does it. In a nut shell it ?s really that easy. More elaborately however, CMake is wonderful for it ?s design to support complex directory hierarchies. When you use CMake it will locate include files, libraries, executables, other optional build directives, then link everything for you. No more need of Makefiles.

CMake basically requires one thing, a CMakeLists.txt file.

Below is the CMakeLists.txt file for our source code.

```
#  cmake;
#
```

```
#  Author(s): Kyoshe Winstone <wkyoshe@gmail.com>
#       Date: 2015-06-18
#
#  University of Strasbourg.
#
cmake_minimum_required(VERSION 2.8.9 FATAL_ERROR)
project(operation)

add_library(operationlib operation.cpp operation.h)

target_include_directories (operationlib PUBLIC CMAKE_CURRENT_SOURCE_DIR).add_
```

Now we want to compile our program. Once run for the first time CMake will configure itself to our system, and place the binaries (our output files) into the directory we called it from. This is where we cd into the build folder, and run

```
        cmake ..
```

The ' .. ' argument is the same as the one used in BASH, it tells CMake to execute from the CMakeLists.txt in the previous directory. This call just produced your Makefile, so let's finish it off with the last command.

```
       make
```

Now if we ls into the build folder we'll see all kinds of stuff.

CMakeCache.txt - Think of this as a configuration file. It has many advanced settings to optimize the build process for large projects. (Such as building the Linux Kernel!)
CMakeFiles (directory) - This has a lot of the stuff CMake uses under the hood to build your project.
cmake-install.cmake - Our install file. We didn't use one in this program.
operation- This is our program
Makefile - remember our last command 'make' ? This is the Makefile produced by Cmake, this is really what it all came down to.
We've Now built a program using CMake.

0.6 DOXYGEN-DOCUMENTATION

Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL, Tcl, and to some extent D

It can generate an on-line documentation browser (in HTML) and an off-line reference manual LaTeX from a set of documented source files.
You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions
You can also use doxygen for creating normal documentation.
First, we need to download and install the doxygen application on our computer to be able to generate the required files.

This part of the report covers the following aspects :

— How to put comments in our code such that doxygen incorporates them in the documentation it generates. This is further detailed in the next section. click here to see the documented source code of our project

— Ways to structure the contents of a comment block such that the output looks good, as explained in section Anatomy of a comment block.
We can use the JavaDoc style, which consist of a C-style comment block starting with two *'s, like this :

```
/**
 * ... text ...
 */
```

Or we can use the Qt style and add an exclamation mark (!) after the opening of a C-style comment block, as shown in this example :

```
/*!
 * ... text ...
 */
```

In both cases the intermediate **'s are optional, so

```
/*!
 ... text ...
*/
```

is also valid.
A third alternative is to use a block of at least two C++ comment lines, where each line starts with an additional slash or an exclamation mark.

— Using CMakeLists.txt to allow the doxygen documentation to be generated with make doc when using cmake.
For this to happen, we add the following lines into the CMakeLists.txt

```
 find_package(Doxygen)
if(DOXYGEN_FOUND)
configure_file(CMAKE_CURRENT_SOURCE_DIR/Doxyfile.in{CMAKE_CURRENT_ BINARY_DIR}/
add_custom_target(doc
DOXYGEN_EXECUTABLE{CMAKE_CURRENT_BINARY_DIR}/Doxyfile
WORKING_DIRECTORY CMAKE_CURRENT_BINARY_DIRCOMMENT"GeneratingAPIdocumentationwi
```

Finally,
At this point i am able to document the simple dummy project codes and use DOXYGEN to produce :

1. An on-line documentation browser (in HTML)
2. An off-line reference manual LaTeX from a set of documented source files.

From the Latex files we can generate a pdf format using the command lines
First type

```
pdflatex
```

then

```
pdflatex myfile
```

0.7 REFFERENCES

i. DOXYGEN MANUAL
ii. WIKIBOOKS
iii. CMAKE TUTORIALS
iv. MAKEFILE TUTORIAL
v. WIKIPEDIA