**GCC C-MAKE FRAMEWORK**

Author : Long Dao | <https://louisvn.com>

1. **Introduce**

* GCC C-Make Framework is designed to manage C/C++ software projects.
* Allows users to create, delete, import, export, compile, run, and export test reports for any project.
* To manage a project, I need you to install some of the following software to use the framework.
  + [**Toolchain GCC**](https://winlibs.com/). This is a compiler that takes care of the process of translating from C code to machine code so that the program you write runs on your PC. Download **GCC version 13.2.0 (with POSIX threads - UCRT) 32bit** for environmental consistency (optional).
  + **Makefile**. This is a replacement for typing complex commands into simple commands on your PC's Terminal control window.
  + [**Cygwin**](https://www.cygwin.com/install.html). If you are a Windows user, you need to install this tool to emulate a Linux environment that makes Makefile and other tools work stably.
  + [**VSCode**](https://code.visualstudio.com/download). This is extremely popular word processing software when you write code. It helps you display colors, command prompt when writing, supports debugging configuration, ...
  + [**Python**](https://www.python.org/downloads/). In Python, a [**Gcovr**](https://gcovr.com/en/stable/) tool allows you to create code coverage reports.

1. **Install GCC**

* To install GCC, go to the following path (Github): [**Click here**](https://github.com/brechtsanders/winlibs_mingw/releases/download/13.2.0-16.0.6-11.0.0-ucrt-r1/winlibs-i686-posix-dwarf-gcc-13.2.0-llvm-16.0.6-mingw-w64ucrt-11.0.0-r1.zip)
* After the download is complete, press **Window + E**. In the File Manager window, go to drive **C**, create a folder called "**Toolchain**".
* Then extract the downloaded file to the newly created folder. After extracting, you will have a path**: C:\Toolchain\mingw32**
* Press **Window + S** and search for "**environment**". You'll see this app:



* Tap select the app. Select the **Advanced** > **Environment variables tab...**
* A new window appears. In the **System variables panel** > **Path** > **Edit...**
* A new window appears. Click **New** and add the following links:
  + **C:\Toolchain\mingw32\bin**
  + **C:\Toolchain\mingw32\i686-w64-mingw32\bin**
* Once the addition is complete, press **OK** one by one to exit.
* Finished! To check if you have installed successfully, press **Window + S** and search for **"cmd**"



* Open the application, type **"gcc -v**" in the window. If you see a message like this, it was successful.



1. **Install VSCode**

* To install VSCode, go to the following path: [**Click here**](https://code.visualstudio.com/download)
* After downloading, click open setup file. Select "**I accept the agreement**" > Select "**Create a desktop icon**" > **Next** > **Install**
* Wait for the installation to be completed and it is successful!
* Next, you need to open the newly installed VSCode application to install the necessary tool packages.
* Press **Ctrl + Shift + X**. An "**Extensions**" tab appears.
* Type in the search bar and download the following packages in turn:
  + **C/C++** ; **C/C++ Themes** ; **C/C++ Extension Pack**
  + **Code Runner**
  + **Doxygen Documentation Generator**
  + **Gdb Syntax**

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

* After the installation is complete, press [ **Ctrl + ,** ] to open the "**Settings**" tab.
* Type "**Code-runner: Run In Terminal**" into the search bar. Then click on the square according to the picture below:



1. **Install Cygwin**

* To install Cygwin, go to the following path: [**Click here**](https://www.cygwin.com/setup-x86_64.exe)
* Like step **2** of GCC installation, go to drive **C**, create a folder called "**Cygwin**".
* Next, copy the downloaded setup file to the folder you just created: **C:\Cygwin**
* Click to open the newly copied application. Select **Next** > **Install from Internet**
* In the new window, enter **"C:\Cygwin**" in "**Root Directory**". Under select **All Users** > **Next**
* In the new window, enter **"C:\Cygwin**" in "**Local Package Directory**" > **Next**
* Select **Use System Proxy Settings** > **Next**
* Click on any 1 link to download the data. (I usually choose the 4th link) > **Next**
* Wait 1 moment until a "**Select Packages**" window appears. At **View** select **Full**
* Search for the following packages and select the latest versions: **make, zip, unzip, sed, cygrunsrv, bc, tree**

Note, click on the square with the down arrow to select the version. For example, the following figure appears:

A screenshot of a computer

Description automatically generated with medium confidence

* After selecting the installation version for all the packages listed above, click  **Next**  > **Next** to proceed with the installation.
* Wait until the installation is complete, click **Finish**.
* Then, you open "**Environments**" similar to step **4+** of the GCC installation and add the following path:
  + **C:\Cygwin\bin**
  + **C:\Cygwin\sbin**
  + **C:\Cygwin\usr\sbin**
* After adding Environments, **right-click** on the newly installed application icon. Select **"Run as administrator**" and wait for the application to appear.
* Type in "**cygserver-config**" > **Enter** >... Type "**yes**". Wait for the run to finish and close the window.
* **Note** : If you have Git installed on your PC. Please move Cygwin's paths above Git's paths in Enviroments.
* Restart your PC - required.

1. **Install Python**

* To install Python, go to the following path: [**Click here**](https://www.python.org/downloads/)
* Click the button "**Download Python ...**" to download the app.
* Press **Window + E**. In the File Manager window, go to drive **C**, create nested folders as follows: "**Toolchain > Python > Python311**". (Substitute"**311**" into your version of Python).
* After downloading, Click open the application. At the settings window, select "**Customize installation**". Next, click **all** the options in "**Optional Features**", especially "**Pip**". Then click "**Next**".
* In the "**Advanced Options**" window also click all options. Then, in the "**Customize install location**" section, enter the path to the previously created folder **"C:\Toolchain\Python\Python311"** and select "**Install**".
* Wait for the installation to complete and press "**Close**".
* After the installation is complete, open **"Path**" under "**Eviroments**". If you don't see the same 2 paths placed at the top (as shown below), copy the exact path on your PC and add it (scroll to the top).

A screenshot of a computer program

Description automatically generated

* Then, open "**cmd**" and type **"python --version**" to check the version of Python. If there is version information printed, you have successfully installed it.

A black text on a white background

Description automatically generated

* Next, at the "**cmd**" window, enter **"pip install gcovr" to install** the Gcovr **tool**. Wait until the tool is successfully installed, you will see the text "Successfully installed gcovr-...". Type **"gcovr --version**" to check its version.



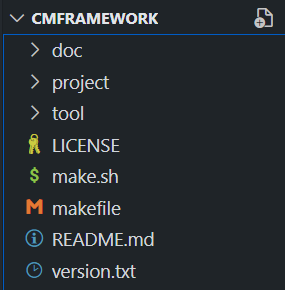
* Note. **Python version 3.11.4** and **Gcovr version 6.0** are running stably on this framework.

1. **Using the Framework**

* You need to download and unzip the Framework before working with it.
* To use the framework, first open the **VSCode** editor .
* To add the Framework to the editor, repeatedly press 2 combinations  **Ctrl + K** and **Ctrl + O**.
* An "**Open Folder"** windowwill appear, select the path to the previouslyextracted **"CMFramework**" folder. Then select “**Select Folder**”.
* [**If**] a window similar to the one below appears, click on the square and click "**Yes**".



* You will see in the "**Explorer**" tab all the items of the newly added framework.
* **Structure of the Framework**



* + **doc** : stores documents for common use for every project. File "**Readme.en.docx**" is also located in this folder.
  + **project**: Manage all your projects. Note, I have provided a sample project "**~temp**" as a foundation for creating other projects. You force not to delete it. If deleted, you will not be able to use any of the features of the framework.
  + **tool** : Includes common libraries, and tools for creating test reports, ...
  + **makefile** : this is a file that manages all the backbone features to ensure the framework works. You also **cannot** customize this file.
  + **make.sh** : This file allows you to run make commands for multiple projects at the same time.
  + **version.txt** : This file records the current version and version history of the framework.
* **Structure of Project in Framework**



Take the example on the sample project "**~temp**" provided. Note that you **may not** edit or delete this project. You need to create a new project and execute on it.

* + **doc** : archives of documents of this project.
  + **Inc** : contains project header files (.**h .hh .hpp**)
  + **dev** : contains the source files of the project developed for testing code coverage (.**c .cc .cpp**)
  + **src** : contains the source files of the project (**.c .cc .cpp .o**)
  + **user\_cfg.mk** : a makefile that allows users to configure paths and set some settings for each specific project.
* **Use Make commands**

First, open a Terminal window on VSCode. Enter  **Terminal**  > **New Terminal**

A screenshot of a phone

Description automatically generated with medium confidence

At the **TERMINAL** tab, enter your requirements. Note, the root path on the terminal should be the path to "**CMFramework**". If this is not the case, use the command **"cd**" to move to.

* + **make setup** : initialize (or re-initialize) the framework, clean up all dependencies of all projects (if any).
  + **make info** : used to print to the screen some information about the project.
  + **make** | **make quick** : These 2 commands allow you to run {**make build run**}. It checks for previous dependencies and only rebuilds files that have been changed.
  + **make force** : This command allows you to run {**make clean build run**}. It forcibly rebuilds all source code files without checking for dependencies.
  + **make clean** : used to delete the output directory, where object files and executable files are stored. It also deletes test reports (if any).
  + **make build** : used to compile files (.**c.cc .cpp**) into object files (**.o**), then bind it to an executable file (**.exe**).
  + **make run**  : used to run executable files (**.exe**) on Terminal.
  + **make debug** : used to debug program on Terminal using **GDB**.
  + **make report** : used to create a test report if you use the "**utest"** library(available in "**tool > lib**") in the C/C++ program to write **test cases**. Or (And) generate a code coverage report (if applicable).
  + **make vsinit** : used to create configuration files in VSCode, helping this software to properly link file paths on **the display interface**. When you move to another project, this command is **automatically** run. You just need to run this command again when you **update** the path or settings in makefile (**user\_cfg.mk**).
  + **make move** : used to move to any project. Automatically add new ones if the project doesn't exist.
    - Example 1: make move.proj1 : will mean move to the project "**proj1**".
    - Example 2:  make move.group1/proj1 : move to project "**proj1"** that is in project group "**group1**". You can nest in multiple groups, using the "**/**" sign to separate.
  + **make remove**: used to delete any project or group. If you delete the current project, it will automatically switch back to the "**~temp**"sample project. The usage is the same as the **"move**" command.
  + **make import zip** : Used to add a new project or group from a shared zip file fromanyone's "**export**" command. The usage is the same as the **"move**" command. However, the variable **"zip**" will indicate the path to the zip file.
    - For example: make import.proj2 zip="path/to/file.zip"
    - Note: the variable "zip" must be written with a "=" sign and path. Do not use white space in between.
  + **make export** : used to package any project or group. As a result, you will get a zip file saved in the "**share**" folder. You can store it or share it with anyone. The usage is the same as the "**move**" command.
  + **Make list** : used to list all the files and folders that are in the project.
  + **make print** : used to print out the values of variables used inside makefile.
    - For example, make print.VAR1.VAR2 : prints out the values of two variables as"**VAR1**" and "**VAR2**".
    - Similarly, if printed multiple variables will be separated by a period"**.**".
    - This statement is only for use by admins or if you are a framework developer.
  + **bash make.sh <options> ::: <projects>** : This command allows users to execute "**make**" commandson multiple projects at once, which speeds up the process of retrieving reports once they have been reliably executed.
    - Example : bash make.sh force report ::: test1 group1/proj1 group1/proj2 : in the above example, we will perform "**make clean build run report**" on projects "**test1**", "**group1/proj1**", and "**group1/proj2**" respectively.
    - Other examples : bash make.sh force report .

or : bash make.sh ::: test1 group1/proj1 .

* **Note**
  + All paths and file names must be written continuously. That is, there is no white space in the middle. You should be mindful when naming any one file or folder.
  + It is not allowed to name source files or header files that coincide with the name of the project. This may cause conflicts when creating output files.
  + In a project, each filename is unique across all directories. You are not allowed to compile 2 identical filenames as it may also have a conflict at the output.
* **Special inputs**
  + You can pass data into your program through special inputs such as definitions (**USER\_DEFS**), arguments (**VAR\_ARGS**), or through environment variables (**USER\_ENVS**).
    - **USER\_DEFS** : A definition must have a "**-D**" flag in front of it.

Ex : -DMAC\_A -DMAC\_A=1 -D”MAC\_A=Hello world” -D”MAC\_A=\”Hello world\””

* + - **VAR\_ARGS** : The arguments are separated by spaces. In case the inside of an argument contains spaces, it needs to enclose **quotation marks**.

Ex : Hello\_world “Hello world” Hello” “world

* + - **USER\_ENVS** : Environment variables need to be defined inside **makefile**. Then **export** those variables. Finally, add the variable name to the **USER\_ENVS**.

Ex : VAR\_ENV := Hello world > export VAR\_ENV > USER\_ENVS += VAR\_ENV

* + In addition, special inputs may contain special ASCII characters such as the following:

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | | |
| **USER\_DEFS** | **VAR\_ARGS** | **USER\_ENVS** |
| \\ | \ | \ | \\ |
| \” | “ | “ | \” |
| \\” | \” | {ERROR} | \\” |
| \n | [ENTER] | \n | \n |
| \t | [TAB] | \t | \t |
| “a b” | a b | a b | “a b” |

1. **Debug**

There are 2 solutions to debug a project after compilation has been completed (.exe file is available).

* **Using the command line (CLI)**
  + Run the "**make debug**" command and perform debugging using **GDB**.
* **Using the VSCode interface (GUI)**
  + Run the "**make vsinit**" command to ensure you have updated all necessary environment variables, definitions, paths to the VSCode interface.
  + On the **VSCode** interface, select the "**Run and Debug** (**Ctrl + Shift + D**)" tab. Then select the "**CMFramework Debug**" task.

A screen shot of a computer

Description automatically generated

* + Finally, click on the "**Play** (**F5**)" button in the image above to perform debugging.
* **Note**
  + You need to understand the commands used for debugging GDB, and/or how to debug on VSCode before using this feature.

1. **Use other tools**

* **Use the Code Testing Tool (utest)**
  + To use this tool, you first need to add the path of this library to your makefile.
  + Open the "**user\_cfg.mk**" file. In the variable "**SRC\_DIRS**" used to scan and retrieve source code files (.c) and the variable "**INC\_DIRS**" used to scan and retrieve header files (.h), you need to add the following path to both variables: "**$(TOOL\_DIRS)/lib**".

A screenshot of a computer program

Description automatically generated

Specifically, **$(TOOL\_DIRS)** is the path to the "**tools**" directory of the Framework. Then"**/lib**" is for accessing the "**lib**" directory inside. The **"\**" sign to join the line below. You can write values that are the same line and separated by "**spaces**".

* + After adding the path, you run the "**make vsinit**" command to update the path to VSCode, thereby being able to prompt you better.
  + Go back to your source file, just include "**utest.h**" and experience.

A screen shot of a computer

Description automatically generated

* + To run tests, you need to create test functions in the given format.

Syntax: **FuncTest( name\_test ) { /\* body \*/ }**

In particular, **"name\_test**" is the name of the function specified by yourself. The upper jaw is also equivalent to :  **void name\_test ( void ) { /\* body \*/ }** . However, writing the correct syntax will make it easier for readers to understand.

A screenshot of a computer

Description automatically generated

In the above example, we have a test function called "**Test\_001**".

* + The purpose of testing is created to check the correctness of the source code. So, to know if your testing is right or wrong, you need to have the test conditions.
  + For example, you want to prove that "1+1=2" and "2+2=4" ... Those are the test conditions. Then, "**UT\_Assert**" is the command that helps you check the conditions.

A screenshot of a computer code

Description automatically generated

In the example above, I have 3 conditions to test. As long as one of the conditions is wrong, the whole test function will result in failure. So, we can say that "**Test\_001**" failed because the 3rd test condition was wrong.

* + After creating the test functions, you need to add these functions to a container to execute it.

A black background with white lines

Description automatically generated

The above 2 macros will create a container, you just need to add the test functions in the syntax: **UT\_AddTest( name\_test, "Brief for name\_test" )**

Where "**name\_test**" is the name of the test function that was created earlier."**Brief for name\_test**" is a title you set yourself.

A computer screen shot of a message

Description automatically generated

If you have multiple test functions, you can simply add more **"UT\_AddTest**" commands. Note that there is **no** "**;**" after these commands. At the same time, this container is not written in any one function. Tests that don't currently want to execute, just discard or note the inside of the container.

* + After adding to the container, you need to make sure that the test functions are declared before using it.

A black background with white lines

Description automatically generated

* + To execute the tests inside the container one by one, you need to use the following command : **UT\_RunTests()**

A screenshot of a computer

Description automatically generated

* + Finally, you run the "**make**" command to compile and run the executable. Then, run **"make report**" to export the result to an html file and view it in a web browser. This html file is saved in the "**doc**" folder of the project.

A screenshot of a computer

Description automatically generated

A screenshot of a test

Description automatically generated

A screenshot of a computer code

Description automatically generated

As you can see, Test Condition 3 was wrong at line 55 in the source code file.

* + Good luck!
* **Use the Code Coverage Measurement (CCOV) tool**
  + Code coverage is an indispensable tool for source code testing, it shows how much coverage of the testing process is on the source code.
  + To run the coverage code, go to "**user\_cfg.mk**", add the variable **"$(DEV\_DIR)**" to the paths of "**SRC\_DIRS**". That allows makefile to scan source code files in the "**dev**" folder.

A screenshot of a computer screen

Description automatically generated

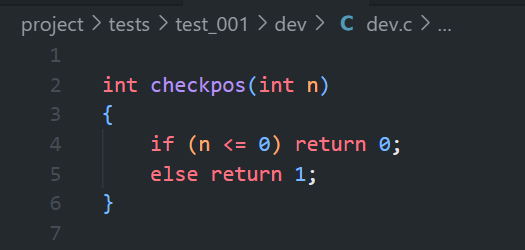
The "**dev**" directory is the directory used to measure CCOV. All functions inside all files in this directory are measured.

* + Next, fix the value of the variable **"RUN\_CCOV**" to "**on**" to allow the use of CCOV.

A close up of a report

Description automatically generated

* + After that, you need to create a source code file inside **"dev**" to write the functions to measure CCOV.



The above example created a "**checkpos**" function to check for positive integers (n > 0) in the "**dev.c**" file located inside the "**dev**" directory.

A computer code with numbers and symbols

Description automatically generated

Then, I created 2 conditions for it to scan both the "**if**" and "**else**"branches inside the **"checkpos**" function. From there, my CCOV index will be completely covered.

* + Finally, you run "**make**" to compile and execute the source code. Then run **"make** **report**" to generate reports including test results and CCOV. They are all stored in the project's "**doc**" folder.

A screen shot of a computer

Description automatically generated

A screenshot of a test

Description automatically generated

A table with numbers and text

Description automatically generated A close-up of a computer code

Description automatically generated

* + Good luck!

=================================================

THE END

Author : Long Dao | <https://louisvn.com>