

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE
AREQUIPA



INGENIERÍA DE SOFTWARE II

Pruebas Unitarias con GTest

Alumna :

- Chullunquía Rosas, Sharon Rossely

Profesor :

- Sarmiento Calisaya, Edgar

10 de noviembre de 2020

Índice

| | |
|---|-----------|
| 1. Introducción | 2 |
| 1.1. GTest: Herramienta para pruebas unitarias en c++ | 2 |
| 1.2. Conceptos Básicos | 2 |
| 1.3. Assertions | 2 |
| 1.3.1. Basic Assertions | 2 |
| 1.3.2. Binary Comparison | 3 |
| 1.3.3. String Comparison | 3 |
| 2. Pruebas Unitarias con GTest | 5 |
| 2.1. Paso 1: Creación de un directorio | 5 |
| 2.2. Paso 2: Agregando archivos fuente y de prueba | 5 |
| 2.3. Paso 3: Agregando googletest | 7 |
| 2.4. Paso 4: Creando el archivo CMakeLists | 8 |
| 2.5. Paso 5: Ejecución de las pruebas | 8 |
| 3. Integración de GTest a Visual Studio Code | 10 |
| 3.1. Paso 1: Instalación de CMake Tools | 10 |
| 3.2. Paso 2: Configuración del kit y del objetivo | 10 |
| 3.3. Paso 3: Ejecución de las pruebas en VS Code | 11 |

1. Introducción

1.1. GTest: Herramienta para pruebas unitarias en c++

[googletest](#) es un framework de prueba desarrollado por el equipo de *Testing Technology* con los requisitos y restricciones específicos de Google en mente. Ya sea que trabaje en Linux, Windows o Mac, si escribe código C++, *googletest* puede ayudarlo. Y admite cualquier tipo de prueba, no solo pruebas unitarias.

Ya sea que trabaje en Linux, Windows o Mac, si escribe código C++, *googletest* puede ayudarlo. Y admite cualquier tipo de prueba, no solo pruebas unitarias.

1.2. Conceptos Básicos

- Cuando usa *googletest*, comienza escribiendo *assertions*, que son declaraciones que comprueban si una condición es verdadera. El resultado de un *assertion* puede ser un éxito, un fracaso no fatal o un fracaso fatal. Si ocurre una falla fatal, aborta la función actual; de lo contrario, el programa continúa normalmente.
- Las pruebas utilizan *assertions* para verificar el comportamiento del código probado. Si una prueba falla o tiene una afirmación fallida, falla; de lo contrario, tiene éxito.
- Un conjunto de pruebas contiene una o varias pruebas. Debe agrupar sus pruebas en conjuntos de pruebas que reflejen la estructura del código probado. Cuando varias pruebas en un conjunto de pruebas necesitan compartir objetos y subrutinas comunes, puede colocarlas en una clase de dispositivo de prueba.
- Un programa de prueba puede contener varios conjuntos de pruebas.

1.3. Assertions

Las *assertions* de *googletest* son macros que se asemejan a llamadas a funciones. Prueba una clase o función haciendo *assertions* sobre su comportamiento. Cuando una *assertion* falla, *googletest* imprime el archivo de origen del *assertion* y la ubicación del número de línea, junto con un mensaje de falla. También puede proporcionar un mensaje de error personalizado que se adjuntará al mensaje de *googletest*.

1.3.1. Basic Assertions

Estas *assertions* hacen pruebas básicas de condición verdadero/falso

| Fatal assertion | Nonfatal assertion | Verifies |
|---------------------------------------|---------------------------------------|------------------------------------|
| <code>ASSERT_TRUE(condition);</code> | <code>EXPECT_TRUE(condition);</code> | <code>condition</code> is true |
| <code>ASSERT_FALSE(condition);</code> | <code>EXPECT_FALSE(condition);</code> | <code>condition</code> is false |

Recuerde, cuando fallan, `ASSERT_*` produce una falla fatal y regresa de la función actual, mientras que `EXPECT_*` produce una falla no fatal, permitiendo que la función continúe ejecutándose. En cualquier caso, una aserción fallida significa que la prueba que la contiene falla.

1.3.2. Binary Comparison

Esta sección describe *assertions* que comparan dos valores.

| Fatal assertion | Nonfatal assertion | Verifies |
|-------------------------------------|-------------------------------------|------------------------------|
| <code>ASSERT_EQ(val1, val2);</code> | <code>EXPECT_EQ(val1, val2);</code> | <code>val1 == val2</code> |
| <code>ASSERT_NE(val1, val2);</code> | <code>EXPECT_NE(val1, val2);</code> | <code>val1 != val2</code> |
| <code>ASSERT_LT(val1, val2);</code> | <code>EXPECT_LT(val1, val2);</code> | <code>val1 < val2</code> |
| <code>ASSERT_LE(val1, val2);</code> | <code>EXPECT_LE(val1, val2);</code> | <code>val1 <= val2</code> |
| <code>ASSERT_GT(val1, val2);</code> | <code>EXPECT_GT(val1, val2);</code> | <code>val1 > val2</code> |
| <code>ASSERT_GE(val1, val2);</code> | <code>EXPECT_GE(val1, val2);</code> | <code>val1 >= val2</code> |

Los argumentos de valor deben ser comparables por el operador de comparación del *assertion* u obtendrá un error del compilador. Solíamos requerir los argumentos para admitir el operador `<<` para transmitir a un ostream, pero esto ya no es necesario. Si `<<` se admite, se llamará para imprimir los argumentos cuando falle el *assertion*; de lo contrario, *googletest* intentará imprimirlos de la mejor manera posible.

1.3.3. String Comparison

Las *assertions* de este grupo comparan dos cadenas en C. Si desea comparar dos objetos de cadena, use `EXPECT_EQ`, `EXPECT_NE`, etc. en su lugar.

| Fatal assertion | Nonfatal assertion | Verifies |
|--|--|--|
| <code>ASSERT_STREQ(str1, str2);</code> | <code>EXPECT_STREQ(str1, str2);</code> | the two C strings have the same content |
| <code>ASSERT_STRNE(str1, str2);</code> | <code>EXPECT_STRNE(str1, str2);</code> | the two C strings have different contents |
| <code>ASSERT_STRCASEEQ(str1, str2);</code> | <code>EXPECT_STRCASEEQ(str1, str2);</code> | the two C strings have the same content, ignoring case |
| <code>ASSERT_STRCASENE(str1, str2);</code> | <code>EXPECT_STRCASENE(str1, str2);</code> | the two C strings have different contents, ignoring case |

Tenga en cuenta que **CASE**.^{en} un nombre de *assertion* significa que se ignora el caso. Un puntero NULL y una cadena vacía se consideran diferentes

2. Pruebas Unitarias con GTest

2.1. Paso 1: Creación de un directorio

Creamos la carpeta *projects* y dentro de ella la carpeta *cpp*, en la ubicación de su preferencia.

```
/home/user/Escritorio/projects/cpp/ # your project lives here
```

y nos ubicamos dentro de la ruta *projects/cpp* .

```
sharon@r2d2:~$ cd Escritorio/  
sharon@r2d2:~/Escritorio$ mkdir projects && cd projects  
sharon@r2d2:~/Escritorio/projects$ mkdir cpp && cd cpp  
sharon@r2d2:~/Escritorio/projects/cpp$
```

Figura 1: En la ruta *projects/cpp*

2.2. Paso 2: Agregando archivos fuente y de prueba

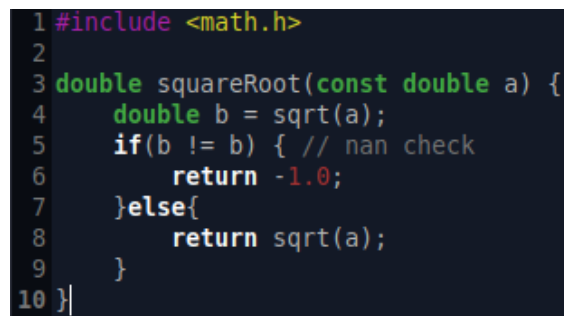
Creamos el archivo *myfunctions.h*, *square.h* y el archivo *mytests.cpp* .

```
1  #ifndef _ADD_H  
2  #define _ADD_H  
3  
4  int add(int a, int b)  
5  {  
6      return a + b;  
7  }  
8  
9  #endif
```

```
1 #ifndef _ADD_H  
2 #define _ADD_H  
3  
4 int add(int a, int b)|  
5 {  
6     return a + b;  
7 }  
8  
9 #endif
```

Figura 2: Archivo *myfunctions.h*

```
1  #include <math.h>
2
3  double squareRoot(const double a) {
4      double b = sqrt(a);
5      if(b != b) { // nan check
6          return -1.0;
7      } else {
8          return sqrt(a);
9      }
10 }
```

A screenshot of a code editor showing the content of the file square.h. The code is identical to the one shown in the previous block, with line numbers 1 through 10 on the left margin. The code defines a function squareRoot that takes a double argument a and returns its square root, with a NaN check. The code is color-coded: #include is purple, double is blue, const is green, and return is red.

```
1 #include <math.h>
2
3 double squareRoot(const double a) {
4     double b = sqrt(a);
5     if(b != b) { // nan check
6         return -1.0;
7     } else {
8         return sqrt(a);
9     }
10 }
```

Figura 3: Archivo *square.h*

```
1  #include <gtest/gtest.h>
2  #include "myfunctions.h"
3  #include "square.h"
4
5  TEST(myfunctions, add)
6  {
7      GTEST_ASSERT_EQ(add(10, 22), 32);
8  }
9
10 TEST(SquareRootTest, PositiveNos) {
11     ASSERT_EQ(6, squareRoot(36.0));
12     ASSERT_EQ(18.0, squareRoot(324.0));
13     ASSERT_EQ(25.4, squareRoot(645.16));
14     ASSERT_EQ(0, squareRoot(0.0));
15 }
16
17 TEST(SquareRootTest, NegativeNos) {
18     ASSERT_EQ(-1.0, squareRoot(-15.0));
19     ASSERT_EQ(-1.0, squareRoot(-0.2));
20 }
```

```
21
22     int main(int argc, char* argv[])
23     {
24         ::testing::InitGoogleTest(&argc, argv);
25         return RUN_ALL_TESTS();
26     }
```

```
1 #include <gtest/gtest.h>
2 #include "myfunctions.h"
3 #include "square.h"
4
5 TEST(myfunctions, add)
6 {
7     GTEST_ASSERT_EQ(add(10, 22), 32);
8 }
9
10 TEST(SquareRootTest, PositiveNos) {
11     ASSERT_EQ(6, squareRoot(36.0));
12     ASSERT_EQ(18.0, squareRoot(324.0));
13     ASSERT_EQ(25.4, squareRoot(645.16));
14     ASSERT_EQ(0, squareRoot(0.0));
15 }
16
17 TEST(SquareRootTest, NegativeNos) {
18     ASSERT_EQ(-1.0, squareRoot(-15.0));
19     ASSERT_EQ(-1.0, squareRoot(-0.2));
20 }
21
22 int main(int argc, char* argv[])
23 {
24     ::testing::InitGoogleTest(&argc, argv);
25     return RUN_ALL_TESTS();
26 }
```

Figura 4: Archivo *mytests.cpp*

Quedando así la estructura de nuestro directorio.

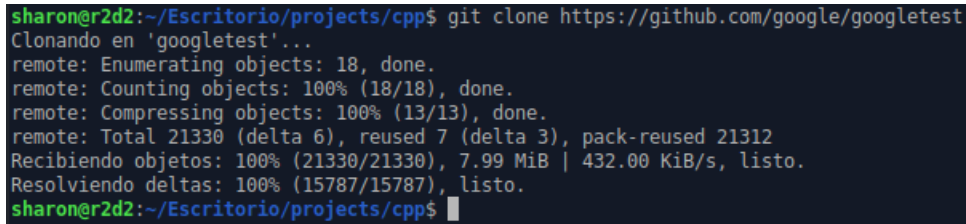
```
└─cpp/
   ├── CMakeLists.txt
   ├── myfunctions.h
   └── mytests.cpp
```

Figura 5: Estructura del directorio *cpp*

2.3. Paso 3: Agregando googletest

Clonamos el repositorio de [googletest](#) dentro de la carpeta *cpp*.


```
git clone https://github.com/google/googletest
```



```
sharon@r2d2:~/Escritorio/projects/cpp$ git clone https://github.com/google/googletest
Clonando en 'googletest'...
remote: Enumerating objects: 18, done.
remote: Counting objects: 100% (18/18), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 21330 (delta 6), reused 7 (delta 3), pack-reused 21312
Recibiendo objetos: 100% (21330/21330), 7.99 MiB | 432.00 KiB/s, listo.
Resolviendo deltas: 100% (15787/15787), listo.
sharon@r2d2:~/Escritorio/projects/cpp$
```

Figura 6: Clonando el repositorio en el directorio *cpp*

2.4. Paso 4: Creando el archivo CMakeLists

Creamos el archivo *CMakeLists.txt* e ingresamos lo siguiente :

```
# version can be different
cmake_minimum_required(VERSION 3.12)

# name of your project
project(my_cpp_project)

# add googletest subdirectory
add_subdirectory(googletest)

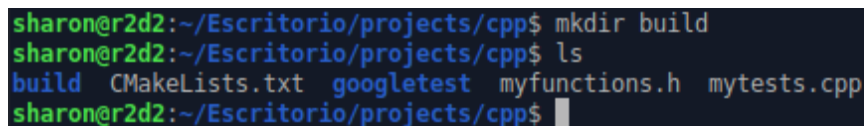
# this is so we can #include <gtest/gtest.h>
include_directories(googletest/include)

# add this executable
add_executable(mytests mytests.cpp)

# link google test to this executable
target_link_libraries(mytests PRIVATE gtest)
```

2.5. Paso 5: Ejecución de las pruebas

En la terminal, dentro de la carpeta *cpp* creamos la carpeta *build*:



```
sharon@r2d2:~/Escritorio/projects/cpp$ mkdir build
sharon@r2d2:~/Escritorio/projects/cpp$ ls
build CMakeLists.txt googletest myfunctions.h mytests.cpp
sharon@r2d2:~/Escritorio/projects/cpp$
```

Ingresamos a la carpeta *build* y corremos los siguientes comandos :

```

sharon@r2d2:~/Escritorio/projects/cpp$ cd build/
sharon@r2d2:~/Escritorio/projects/cpp/build$ cmake ..
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found PythonInterp: /usr/bin/python3.8 (found version "3.8.5")
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Failed
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /home/sharon/Escritorio/projects/cpp/build

```

Figura 7: Ejecutando el comando *cmake ..*

```

sharon@r2d2:~/Escritorio/projects/cpp/build$ make
Scanning dependencies of target gtest
[ 10%] Building CXX object googletest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
[ 20%] Linking CXX static library ../../lib/libgtest.a
[ 20%] Built target gtest
Scanning dependencies of target mytests
[ 30%] Building CXX object CMakeFiles/mytests.dir/mytests.cpp.o
[ 40%] Linking CXX executable mytests
[ 40%] Built target mytests
Scanning dependencies of target gmock
[ 50%] Building CXX object googletest/googlemock/CMakeFiles/gmock.dir/src/gmock-all.cc.o
[ 60%] Linking CXX static library ../../lib/libgmock.a
[ 60%] Built target gmock
Scanning dependencies of target gmock_main
[ 70%] Building CXX object googletest/googlemock/CMakeFiles/gmock_main.dir/src/gmock_main.cc.o
[ 80%] Linking CXX static library ../../lib/libgmock_main.a
[ 80%] Built target gmock_main
Scanning dependencies of target gtest_main
[ 90%] Building CXX object googletest/googletest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o
[100%] Linking CXX static library ../../lib/libgtest_main.a
[100%] Built target gtest_main
sharon@r2d2:~/Escritorio/projects/cpp/build$ ./mytests
[=====] Running 3 tests from 2 test suites.
[-----] Global test environment set-up.
[-----] 1 test from myfunctions
[ RUN      ] myfunctions.add
[ OK       ] myfunctions.add (0 ms)
[-----] 1 test from myfunctions (0 ms total)

[-----] 2 tests from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
[ OK       ] SquareRootTest.PositiveNos (0 ms)
[ RUN      ] SquareRootTest.NegativeNos
[ OK       ] SquareRootTest.NegativeNos (0 ms)
[-----] 2 tests from SquareRootTest (0 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 2 test suites ran. (0 ms total)
[ PASSED  ] 3 tests.

```

Figura 8: Ejecutando el comando *make* y el ejecutable *mytests*

3. Integración de GTest a Visual Studio Code

3.1. Paso 1: Instalación de CMake Tools

En Marketplace de VS Code buscamos la extensión *CMake Tools* e instalamos.

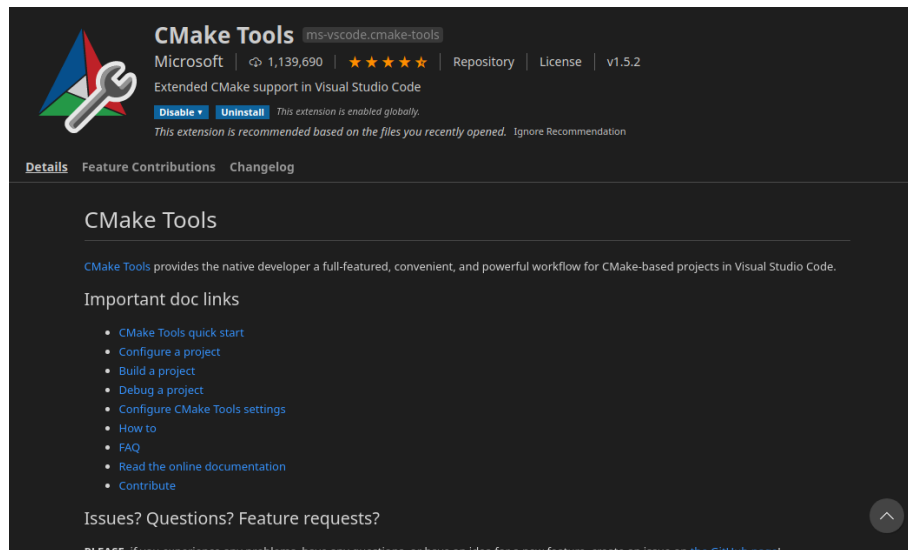


Figura 9: Extensión CMake Tools en VS Code

3.2. Paso 2: Configuración del kit y del objetivo

En la barra inferior, hacemos click en *No se ha seleccionado ningún kit.* y seleccionamos *Clang 10*. Luego, cambiamos el objetivo actual en el que esta (*[all]*), y en vez de *all* colocamos *mytests*, nuestro ejecutable:

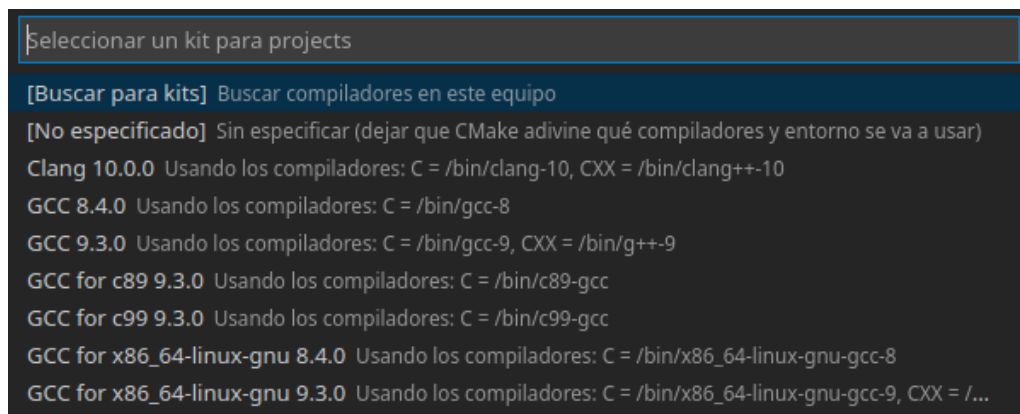


Figura 10: Cambiando a *Clang 10*



Figura 11: Barra inferior de VS Code sin configuración

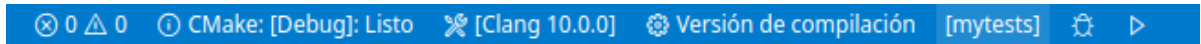


Figura 12: Barra inferior de VS Code ya configurada

3.3. Paso 3: Ejecución de las pruebas en VS Code

Luego, hacemos click en el triángulo que se encuentra en el lado derecho de la barra inferior.

Si nos aparece lo siguiente :

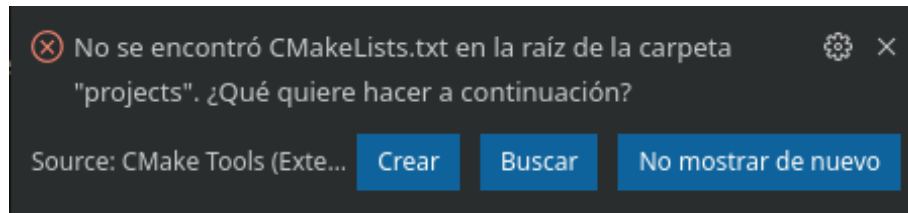
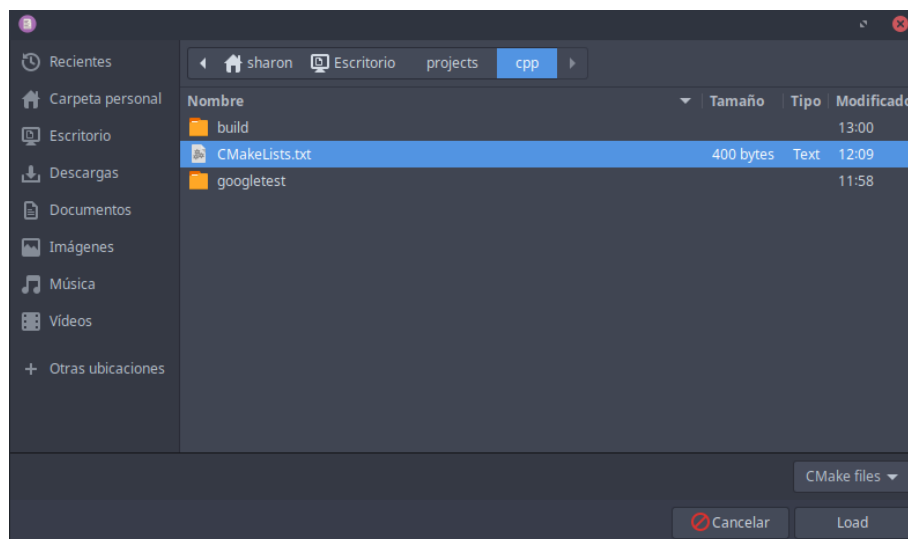
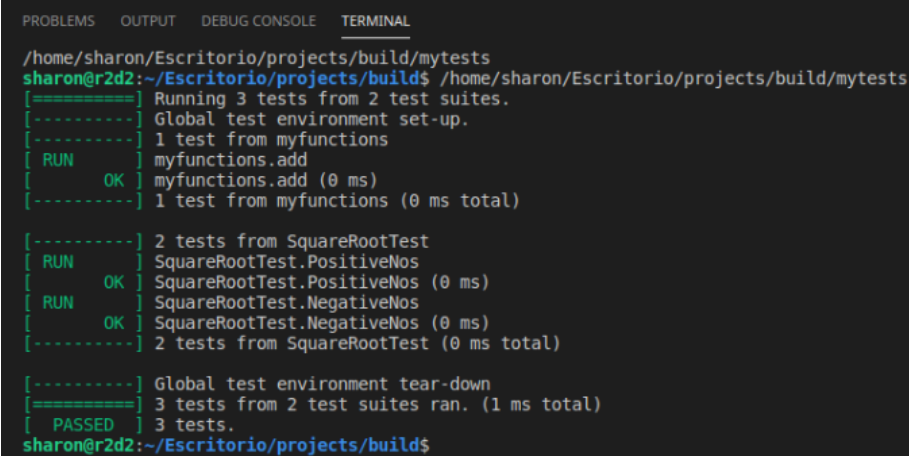


Figura 13: Aviso de CMake Tools

Hacemos click en *Buscar*, buscamos nuestro archivo *CMakeLists.txt* en la carpeta *cpp* y hacemos click en *load*.

Figura 14: Ubicando el archivo *CMakeLists.txt*

Nuevamente hacemos click en el triángulo de la barra inferior , se ejecutarán las pruebas y al final veremos el reporte de googletest.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
/home/sharon/Escritorio/projects/build/mytests
sharon@r2d2:~/Escritorio/projects/build$ /home/sharon/Escritorio/projects/build/mytests
[=====] Running 3 tests from 2 test suites.
[-----] Global test environment set-up.
[-----] 1 test from myfunctions
[ RUN      ] myfunctions.add
[ OK       ] myfunctions.add (0 ms)
[-----] 1 test from myfunctions (0 ms total)

[-----] 2 tests from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
[ OK       ] SquareRootTest.PositiveNos (0 ms)
[ RUN      ] SquareRootTest.NegativeNos
[ OK       ] SquareRootTest.NegativeNos (0 ms)
[-----] 2 tests from SquareRootTest (0 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 2 test suites ran. (1 ms total)
[ PASSED  ] 3 tests.
sharon@r2d2:~/Escritorio/projects/build$
```

Figura 15: Reporte de googletest

Referencias

- [1] “GoogleTest a c++ testing framework by google.” <https://github.com/google/googletest>.
- [2] ”Googletest Primer”<https://github.com/google/googletest/blob/master/googletest/docs/primer.md>.