

# MANUAL PARA USAR GOOGLE TEST

## Contenido:

1. Introducción
2. Que es un Unit Test ?
3. Criterios para Unit Test
4. Configuración Google Test con el cmake
5. Ejemplo Test

## 1. Introducción

El presente Documento contendrá el manejo de GoogleTest para realizar nuestras debidas pruebas.

Para tener el google test, seguir los pasos del manual del Cmake.

## 2. Que es un Unit Test ?

Un unit test es un método que prueba una unidad de código. Al hablar de una unidad de código nos referimos a un requerimiento. Muchos desarrolladores tienen su propio concepto de lo que es una prueba unitaria; sin embargo, la gran mayoría coincide en que una prueba unitaria tiene las siguientes características:

- **Prueba solamente pequeñas cantidades de código:** Solamente prueba el código del requerimiento específico.
- **Se aísla de otro código y de otros desarrolladores:** El unit test prueba exclusivamente el código relacionado con el requerimiento y no interfiere con el trabajo hecho por otros desarrolladores.
- **Solamente se prueban los endpoints públicos:** Esto principalmente porque los disparadores de los métodos privados son métodos públicos por lo tanto se abarca el código de los métodos privados dentro de las pruebas.
- **Los resultados son automatizados:** Cuando ejecutamos las pruebas lo podemos hacer de forma individual o de forma grupal. Estas pruebas las hace el motor de prueba y los resultados de los mismos deben de ser precisos con respecto a cada prueba unitaria desarrollada.

- **Repetible y predecible:** No importa el orden y las veces que se repita la prueba, el resultado siempre debe de ser el mismo.
- **Son rápidos de desarrollar:** Contrariamente a lo que piensan los desarrolladores → que el desarrollo de pruebas unitarias quita tiempo – los unit test por lo general deben de ser simples y rápidos de desarrollar. Difícilmente una prueba unitaria deba de tomar más de cinco minutos en su desarrollo.

### 3. Criterios para Unit Test

- Diseño de clases antes de implementar código, para que así sea más fácilmente testeable.
- Ayudan a la hora de **refactorizar el código**
- Documentación de nuestro código.
- Con las pruebas unitarias y un diseño más modular de nuestra aplicación también conseguimos un **código más desacoplado**.
- **Los errores están más controlados** y para arreglar nuevos bugs siempre podemos crear un nuevo test y hacer que pase en nuestro código.

Desarrollar los unit tests puede parecer un tiempo malgastado en el desarrollo de una aplicación, pero viendo todo lo que nos aportan estas pruebas unitarias, nos ayudan a crear una aplicación más robusta, bien documentada y con código más reutilizable.

## Tipos de aserciones:

### Assertions Condicionales:

Comparacion dada una condición se espera recibir un valor esperado True/False.

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

### Assertions Comparación Binaria:

Se realiza la comparación dado dos valores numéricos.

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

### Assertions Comparación Strings:

Se realiza la comparación dado dos valores de tipo string.

| 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 content                |
| <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 content, ignoring case |

### Exception Assertions :

Verifica que un pedazo de código lanza (o no lanza) una excepción de un tipo dado:

| Fatal assertion   | Nonfatal assertion                                      | Verifies   |
|---|---|--|
| <code>ASSERT_THROW( statement, exception_type );</code> | <code>EXPECT_THROW( statement, exception_type );</code> | <i>statement</i> throws an exception of the given type |
| <code>ASSERT_ANY_THROW( statement );</code>             | <code>EXPECT_ANY_THROW( statement );</code>             | <i>statement</i> throws an exception of any type       |
| <code>ASSERT_NO_THROW( statement );</code>              | <code>EXPECT_NO_THROW( statement );</code>              | <i>statement</i> doesn't throw any exception           |

## 4. Configuración de Google Test en el CMAKE

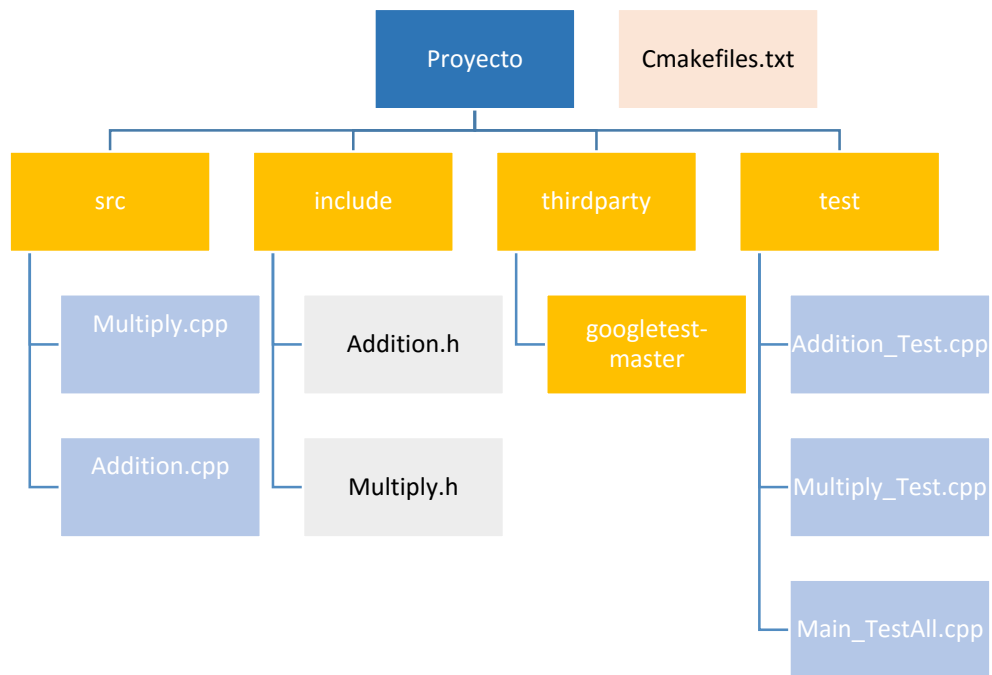
Descargar desde Github el repositorio de GoogleTest:

<https://github.com/google/googletest>

Una vez clonado o descargado el googletest nos dirigimos a su directorio hasta estar a su raíz a través de la terminal de Ubuntu y posteriormente lo descomprimos en nuestro directorio llamado thirdparty.

## 5. Unit Tests con Google Tests

Para realizar unit testing con google tests usando cmake, presentaremos un ejemplo con la siguiente jerarquía.



Donde tenemos en el directorio src:

```
Addition.cpp
#include "Addition.h"
int Addition::twoValues(const int x, const int y)
{
    return x + y;
}
```

### **Multiply.cpp**

```
#include "Multiply.h"
int Multiply::twoValues(const int x, const int y)
{
    return x * y;
}
int Multiply::threeValues(const int x, const int y, const int z)
{
    return x * y * z;
}
```

En el directorio Include:

### **Addition.h**

```
#ifndef _ADDITION_HPP_
#define _ADDITION_HPP_
class Addition
{
public:
    static int twoValues(const int x, const int y);
};

#endif
```

### **Multiply.h**

```
#ifndef _MULTIPLY_HPP_
#define _MULTIPLY_HPP_
class Multiply
{
public:
    static int twoValues(const int x, const int y);
    static int threeValues(const int x, const int y, const int z);
};

#endif
```

En el directorio test:

### **Addition\_Test.cpp**

```
#include <limits.h>
#include "gtest/gtest.h"
#include "Addition.h"

class AdditionTest : public ::testing::Test {
protected:
    virtual void SetUp() {
    }
    virtual void TearDown()
    {
    }
};

TEST_F(AdditionTest, twoValues) {
    const int x = 4;
```

```

    const int y = 5;
    Addition addition;
    EXPECT_EQ(9, addition.twoValues(x,y));
    EXPECT_EQ(5, addition.twoValues(2,3));
}

```

### **Multiply\_Test.cpp**

```

#include <limits.h>
#include "gtest/gtest.h"
#include "Multiply.h"

class MultiplyTest : public ::testing::Test {

protected:

    virtual void SetUp() {
    }
    virtual void TearDown() {

    }
};

TEST_F(MultiplyTest, twoValues) {
    const int x = 4;
    const int y = 5;
    Multiply multiply;
    //EXPECT_EQ(30, multiply.threeValues(2,3,5));
    ASSERT_EQ(20, multiply.twoValues(x,y));
    ASSERT_EQ(6, multiply.twoValues(2,3));
    ASSERT_EQ(34, multiply.twoValues(2,8));
    ASSERT_EQ(44, multiply.twoValues(5,8));
}

```

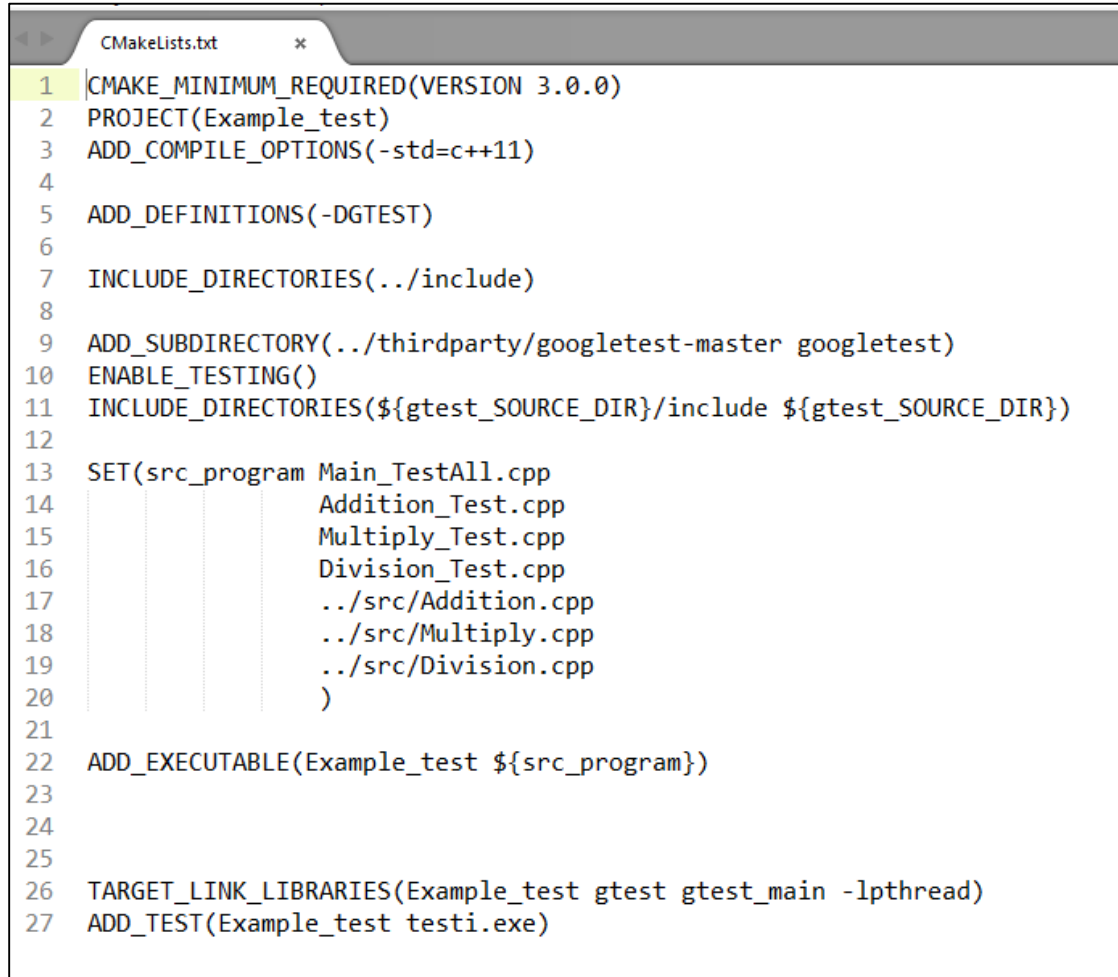
### **Main\_TestAll.cpp**

```

#include <limits.h>
#include "gtest/gtest.h"
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

## El Archivo CmakeList.txt

A screenshot of a code editor window titled 'CMakeLists.txt'. The editor shows a CMake script with 27 lines of code. Line 1 is highlighted in yellow. The code defines a project named 'Example\_test', sets C++11 as the standard, adds definitions for 'DGTEST', includes directories for 'googletest' and local source files, sets source files for a program named 'Main\_TestAll', and links the 'gtest' library. It also defines a test named 'testi.exe'.

```
1 CMAKE_MINIMUM_REQUIRED(VERSION 3.0.0)
2 PROJECT(Example_test)
3 ADD_COMPILE_OPTIONS(-std=c++11)
4
5 ADD_DEFINITIONS(-DGTEST)
6
7 INCLUDE_DIRECTORIES(..../include)
8
9 ADD_SUBDIRECTORY(..../thirdparty/googletest-master googletest)
10 ENABLE_TESTING()
11 INCLUDE_DIRECTORIES(${gtest_SOURCE_DIR}/include ${gtest_SOURCE_DIR})
12
13 SET(src_program Main_TestAll.cpp
14      Addition_Test.cpp
15      Multiply_Test.cpp
16      Division_Test.cpp
17      ../src/Addition.cpp
18      ../src/Multiply.cpp
19      ../src/Division.cpp
20      )
21
22 ADD_EXECUTABLE(Example_test ${src_program})
23
24
25
26 TARGET_LINK_LIBRARIES(Example_test gtest gtest_main -lpthread)
27 ADD_TEST(Example_test testi.exe)
```

El ejemplo se lo puede descargar desde el repositorio [DevInt23Test/pruebatest](#).