9/10/2017

Jhonny Horacio Crespo

MODULO DeVINT 23

Guía de uso para CMake

Proyecto Newbie

Contenido

[1. QUE ES CMAKE 2](#_Toc492854985)

[2. EL PROCESO DE CMAKE 2](#_Toc492854986)

[La Fase de Configuración 2](#_Toc492854987)

[La Fase de Generación 3](#_Toc492854988)

[El proceso de Compilación de C/C++ 4](#_Toc492854989)

[3. REQUISITOS 5](#_Toc492854990)

[4. INSTALACION DE CMAKE 5](#_Toc492854991)

[5. LA ESTRUCTURA DE DIRECTORIOS DEL PROYECTO 5](#_Toc492854992)

[6. CONFIGURACION MINIMA 7](#_Toc492854993)

[7. EL ARCHIVO CMAKELISTS DEL PROYECTO 8](#_Toc492854994)

[8. GENERAR EL BUILD DEL PROYECTO 10](#_Toc492854995)

[9. AGREGAR, EDITAR Y ELIMINAR HEADERS Y SOURCE FILES 13](#_Toc492854996)

[Source Files (Archivos .cpp) 13](#_Toc492854997)

[Header Files (Archivos .h) 14](#_Toc492854998)

[10. AGREGAR BIBLIOTECAS ESTATICAS AL PROYECTO 15](#_Toc492854999)

[Bibliotecas de Terceros (Thirdparty) 15](#_Toc492855000)

[Bibliotecas Propias 15](#_Toc492855001)

[11. GOOGLE TESTS 18](#_Toc492855002)

[12. Instalación de Google Tests 18](#_Toc492855003)

[13. Unit Tests con Google Tests 19](#_Toc492855004)

# QUE ES CMAKE

Cmake es un conjunto de herramientas open-source y multiplataforma diseñadas para construir, testear y empaquetar software (CMake, CTest, CPack y CDash). Cmake es la herramienta usada para controlar el proceso de compilación usando archivos de configuración independientes del compilador y generar makefiles nativos y workspaces que pueden ser usados en el entorno de preferencia.

Cuando cmake fue desarrollado, la práctica normal para un proyecto era tener scripts de configuración y makefiles para Unix y archivos de visual studio para Windows. Este problema hacia el desarrollo multiplataforma muy problemático, la simple acción de agregar nuevos source files a un proyecto era tedioso. El objetivo de los desarrolladores era tener un build system unificado que funcione en todas las plataformas.

# EL PROCESO DE CMAKE

Cmake tiene dos fases principales. El primero es la fase de configuración, en el cual cmake procesa todos los datos de entrada y crea una representación interna del build que se construirá. La siguiente fase es la “fase de generación”, en esta fase los archivos del build son creados.

## La Fase de Configuración

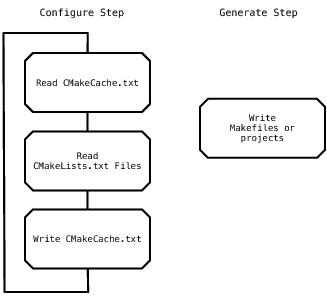
La primera vez que ejecutamos cmake para un generar un build, crea un archivo CMakeCache.txt que almacena todas las variables persistentes para ese build y que luego están disponibles para cmake cuando vuelva a ejecutarse.

Durante la fase de configuración, cmake lee el archivo CMakeCache.txt para ver si ya se ejecutado anteriormente. Luego lee el archivo CmakeLists.txt y ejecuta las instrucciones. Cmake tiene un objeto de C++ para cada comando que puede ser usado en el lenguaje de CMake. En efecto, todo el lenguaje de CMake es implementado como llamadas a comandos. El parser de cmake simplemente convierte los archivos de cmake a llamadas con sus respectivos argumentos.

La fase de configuración esencialmente ejecuta el código cmake provisto. Luego de que todo el código es ejecutado. Cmake creará un archivo CmakeCache.txt que contendrá toda la información para crear los archivos para el build de acuerdo al generador especificado.

## La Fase de Generación

Esta fase incluye todo lo que sucede cuando se genera el build desde los project files creados por cmake, es decir, cuando ejecutamos make y se crea el build.



Solo es necesario un archivo de configuración, llamado CMakeLists.txt, para generar los archivos de configuración del build.

Cmake está diseñado para trabajar con árboles de directorios complejos y aplicaciones que dependen de muchas librerías. Por ejemplo, Cmake puede trabajar con proyectos que consisten de muchas librerías que a su vez pueden contener muchos directorios. También puede manejar situaciones donde se deben construir ejecutables para generar código que luego es compilado y enlazado a la aplicación final.

El proceso de creación es controlado creando uno o más archivos CMakeLists.txt en cada directorio.

CMake no compila o hace el proceso de enlazamiento, usa un generador para crear los archivos de configuración que usa make u otro build-system.

# REQUISITOS

Antes de hacer uso de esta guía, asegúrese de haber seguido todos los pasos del manual de preparación del entorno de desarrollo.

Las versiones de sistema operativo, compilador y cmake con las que se trabajaron al momento de la redacción de este documento se detallan a continuación:

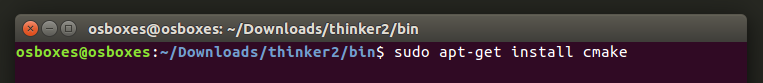
* Sistema Operativo: Ubuntu 16.04 LTS o Ubuntu Mate 16.04.3 LTS
* Compilador g++ versión 5.4.0
* Cake versión 3.5.1

No se asegura el funcionamiento correcto en otras versiones que no sean las especificadas anteriormente.

# INSTALACION DE CMAKE

El proceso de instalación es sencillo, basta con ejecutar el siguiente comando en la consola de Ubuntu:

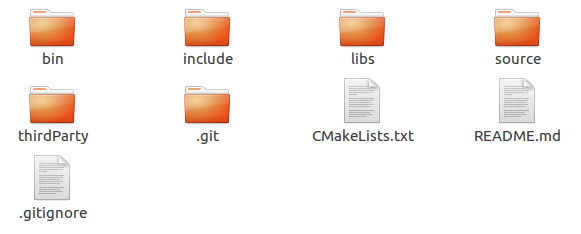
**sudo apt-get install cmake**



Con esto ya tiene cmake instalado en su sistema operativo y está listo para usarse.

# LA ESTRUCTURA DE DIRECTORIOS DEL PROYECTO

Se hará uso de la siguiente estructura de directorios para organizar el proyecto:



A continuación, se describe el uso que se dará a cada directorio:

* **bin**: Este directorio está destinado a contener los archivos que genera cmake. Usted no debe guardar ningún archivo en este directorio.
* **include**: Este directorio está destinado a almacenar los header files (archivos con extensión .h) del proyecto.
* **sources**: Este directorio está destinado a almacenar los source files (archivos con extensión .cpp) del proyecto.
* **libs**: Este directorio está destinado a almacenar las bibliotecas creadas por uno mismo. Ejm: optionparser, thinker, etc. Debe tomar en cuenta que el directorio que contenga su biblioteca, debera seguir esta misma estructura de directorios para organizar sus archivos, puede tomar como ejemplo las bibliotecas optionparser y thinker.
* **thirdParty**: Este directorio está destinado a almacenar las bibliotecas de terceros. Ejm: Boost, binding-cpp, curl, jsonlib, etc.
* **CMakeLists.txt**: Este es el archivo que usa cmake para generar los archivos de configuración del build del proyecto. Deberá editarse, cada vez que se agregue/elimine un source file, se cree una librería propia o se agregue una librería externa que deba ser construida usando cmake.

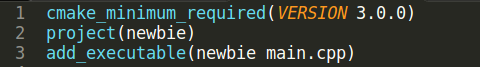
# CONFIGURACION MINIMA

Para hacer uso de cmake es necesario un archivo CmakeLists.txt. Este archivo es el encargado de especificar toda la información que necesita cmake para generar los archivos de configuración del entorno:

* Que source files que se compilaran
* Las bibliotecas que se enlazaran al build
* El nombre del ejecutable a generar y que archivos usara como source files
* Y otros parámetros y variables adicionales

Es importante aclarar que cmake no es case-sensitive por lo que es permitido escribir las instrucciones tanto en mayúsculas como en minúsculas.

Un archivo CmakeLists.txt debe tener al menos las siguientes directivas:



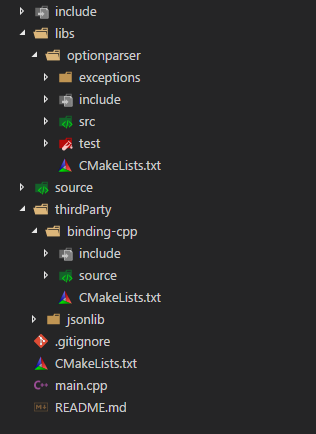
A continuación, se detalla el significado de estas directivas:

* **cmake\_minimum\_required:** Especifica la versión mínima requerida para ejecutar las instrucciones correctamente. Debemos considerar que el comportamiento de cmake puede variar entre versiones.
* **project:** Especifica el nombre del proyecto. Este nombre se utiliza para generar un conjunto de variables que identifican a los distintos directorios.
* **add\_executable:** Crea un ejecutable usando los ficheros especificados como sources. En este ejemplo queremos crear el ejecutable newbie, usando como único source al fichero main.cpp

# EL ARCHIVO CMAKELISTS DEL PROYECTO

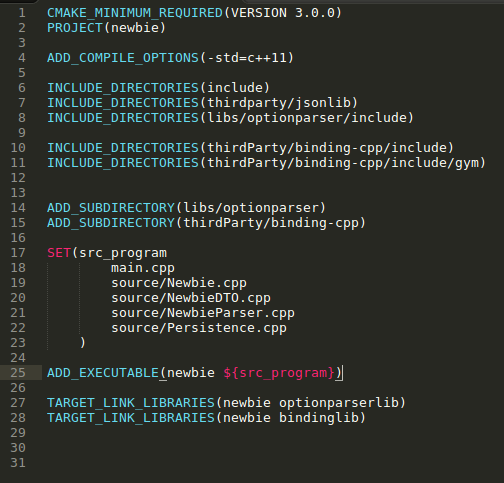
Toda la información que contiene CMakeLists.txt puede ir concentrada en un único fichero por proyecto, sin embargo, es recomendable dividirla en varios ficheros. Es decir, si tenemos un proyecto dividido de forma jerárquica en varios directorios, podemos crear un fichero CMakeLists.txt para cada directorio relevante, y hacer que un CMakeLists.txt principal llame a los otros.

Tomamos como ejemplo el proyecto thinker2 y podemos ver que todos los directorios relevantes contienen un fichero CMakeLists.txt



Esto nos permite generar el build de los componentes del sistema de forma independiente para probarlos.

A continuación, se muestra el archivo CMakeLists.txt principal del proyecto:

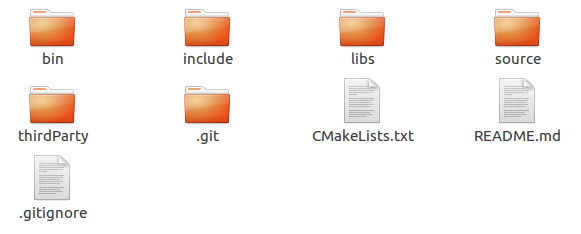


Dónde:

* **CMAKE\_MINIMUM\_REQUIRED:** Especifica la versión mínima de cmake que requiere el proyecto para que el build se cree correctamente.
* **PROJECT:** Es el nombre del proyecto
* **ADD\_COMPILE\_OPTIONS:** Especifica el estándar a usar al compilar
* **INCLUDE\_DIRECTORIES:** Especifica los directorios donde se encuentran los archivos header.
* **ADD\_SUBDIRECTORY:** Especifica los directorios que deben construirse primero antes de crear el ejecutable. Estos directorios deben contener un archivo CMakeLists.txt
* **SET:** Define una variable **src\_program** con la lista de los sources que se usaran para construir el ejecutable.
* **ADD\_EXECUTABLE:** Indica a cmake que queremos construir un ejecutable llamado **newbie** usando como sources la lista que contiene la variable **${src\_program}**
* **TARGET\_LINK\_LIBRARIES:** Indica a cmake que al momento de compilar enlace las librerías **optionparserlib** y **bindinglib** al ejecutable.

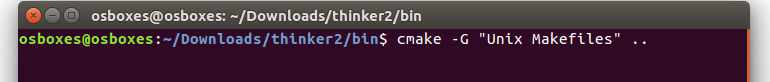
# GENERAR EL BUILD DEL PROYECTO

Para crear el build del proyecto, primero hay que crear un directorio con el nombre de “bin”

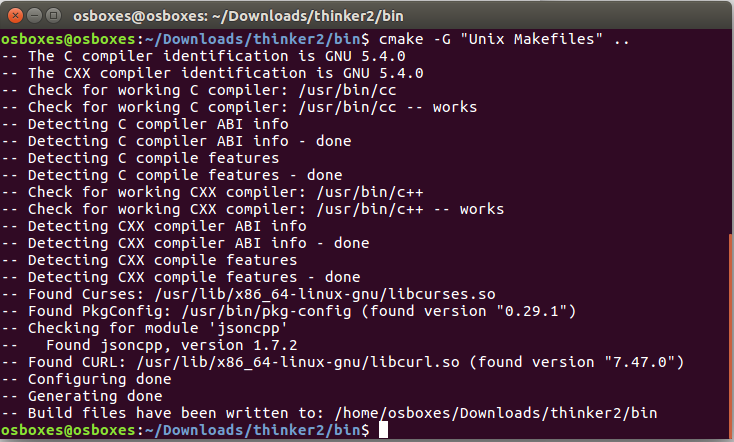


Y usando la consola dirigirnos hasta ese directorio y una vez ahí ejecutar el siguiente comando:

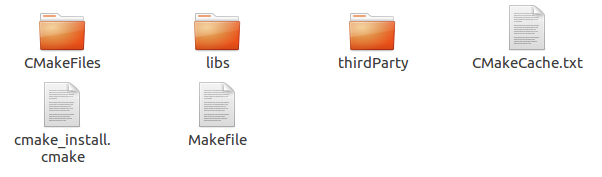
**cmake -G “Unix Makefiles” ..**



Si todo ha ido correctamente deberá ver en consola un resultado parecido al de la siguiente imagen, sin ningún mensaje de error:

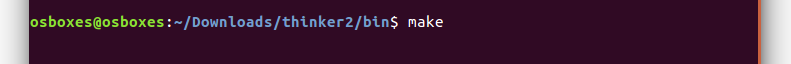


Este comando creara los archivos de configuración en el directorio bin para crear el build del proyecto usando make. En la siguiente imagen se puede ver los archivos que genera cmake al ejecutar el comando.

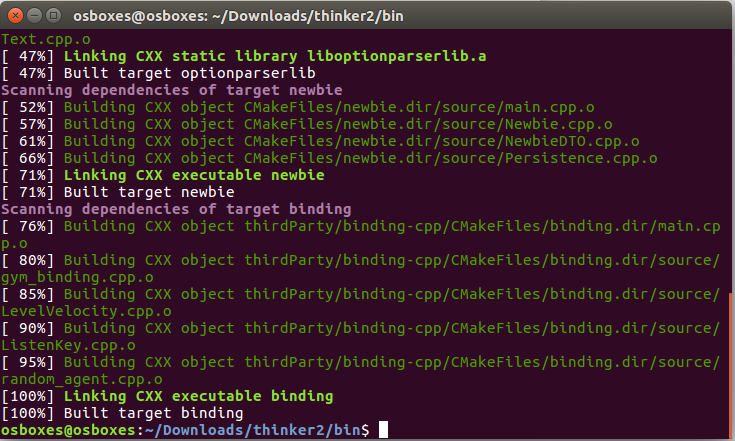


Para crear el build del proyecto (el ejecutable) deberá ejecutar el siguiente comando en la consola:

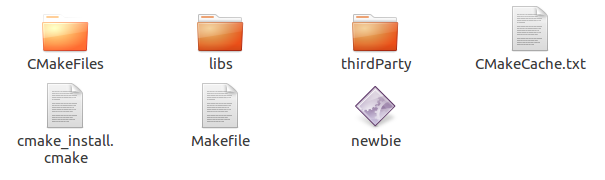
**make**



Si todo ha ido correctamente, deberá ver en la consola un resultado parecido al de la siguiente imagen, sin ningún mensaje de error:



Una vez hecho esto, podrá ver el archivo ejecutable “**newbie**” en el directorio bin:



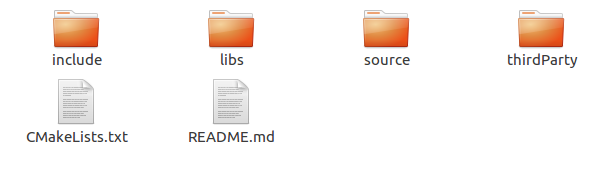
Puede usar este ejecutable para iniciar el programa.

* **En caso de que necesite volver a generar el build, primero debe eliminar todo el contenido del directorio bin y volver a ejecutar los comando descritos anteriormente.**

# AGREGAR, EDITAR Y ELIMINAR HEADERS Y SOURCE FILES

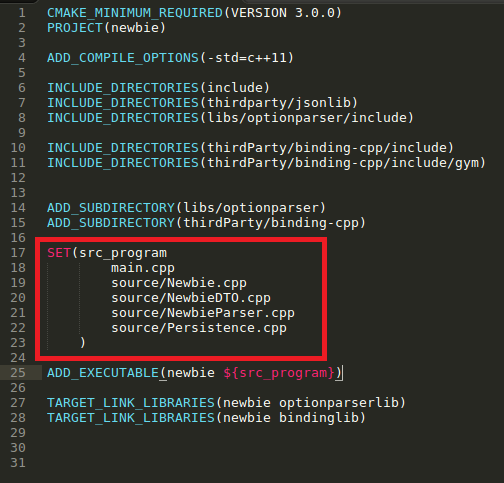
## Source Files (Archivos .cpp)

En el caso que haya creado uno o más source files (archivos .cpp), deberá ponerlos en el directorio “source” del directorio thinker2.



Luego deberá actualizar el archivo CMakeLists.txt agregando la ruta de su source file a la lista de source file. Una vez hecho esto ya podrá generar el build.

Si el directorio bin se encuentra con archivos, deberá eliminarlos todos antes de [generar el build](#_CREAR_EL_BUILD) nuevamente.

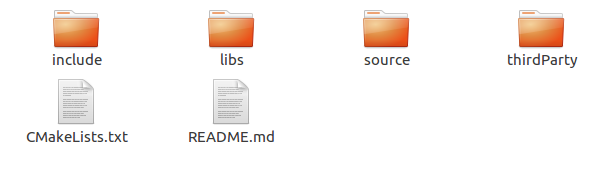


En caso de que haya editado un source file, no es necesario actualizar el archivo CMakelists.txt, simplemente [genere el build](#_GENERAR_EL_BUILD).

En caso de que haya eliminar un source file, debe actualizar el archivo CMakeLists.txt y quitar de la lista de sources al archivo eliminado.

## Header Files (Archivos .h)

En el caso que haya creado uno o más header files (archivos .h) es suficiente con ponerlos en el directorio “include” y [generar el build](#_CREAR_EL_BUILD) sin necesidad de editar el archivo CMakeListst.txt.



En caso de que haya editado un header file, no es necesario actualizar nada del archivo CMakeLists.txt, simplemente [genere el build](#_CREAR_EL_BUILD).

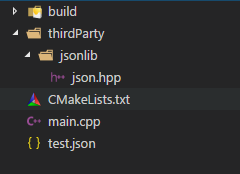
En caso de que haya eliminado un archivo header, no es necesario actualizar nada del archivo CMakeListst.txt.

# AGREGAR BIBLIOTECAS ESTATICAS AL PROYECTO

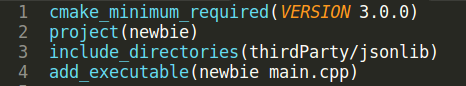
## Bibliotecas de Terceros (Thirdparty)

En caso de que necesite agregar bibliotecas de terceros (Ejm: Boost, jsonlib, etc) deberá agregar el directorio de la biblioteca al directorio thirdParty del proyecto principal y actualizar el archivo CMakeLists.txt.

Para este ejemplo se hara uso de la librería jsonlib. Como puede ver la librería consta simplemente de un archivo json.hpp.



Entonces en el archivo CMakeLists.txt debe decirle a cmake donde puede encontrar el archivo json.hpp, esto se hace usando la directiva **include\_directories** y pasándole como argumento la dirección donde se encuentra el archivo json.hpp con respecto al archivo CMakeLists.txt.



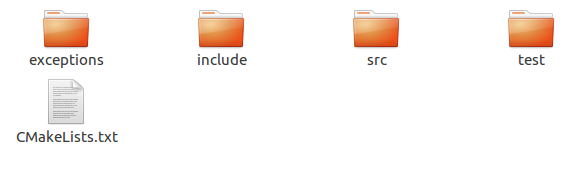
Es importante aclarar que al momento de importar la biblioteca, la instrucción **#include** deberá hacerse usando angle brackets (< >) y no comillas dobles (“ ”).



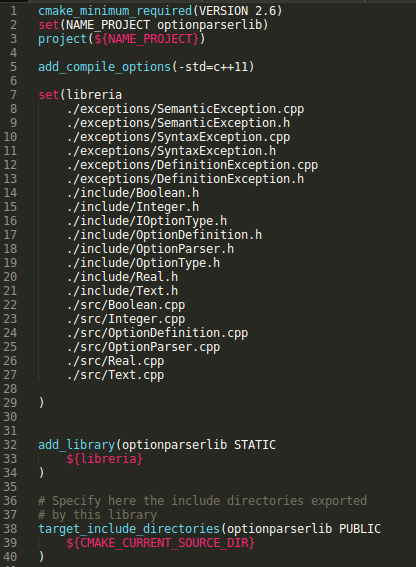
## Bibliotecas Propias

En caso de que necesite crear su propia biblioteca deberá crear un nuevo directorio con el nombre de su biblioteca (Ejm: optionparser) en el directorio libs y seguir la misma [estructura de directorios](#estructura_directorios) que usa el proyecto principal.

Puede tomar como referencia la biblioteca optionparser:



El directorio de su biblioteca deberá tener su propio archivo CMakeLists.txt con la configuración para generar la biblioteca de forma independiente al proyecto principal. A continuación, se detalla los comandos que debe tener el archivo CMakeLists.txt para generar una biblioteca estática correctamente:



Donde:

* **CMAKE\_MINIMUM\_REQUIRED:** Especifica la versión de cmake mínima requerida para ejecutar este archivo sin problemas.
* **SET:** Declara y define una variable NAME\_PROJECT con el nombre optionparserlib que puede ser usado luego en el archivo.
* **PROJECT:** Define el nombre del Proyecto usando la variable NAME\_PROJECT.
* **ADD\_COMPILE\_OPTIONS:** Especifica al compilador que use el estándar 11 de C++ para compilar los archivos.
* **SET:** Define y declara una variable librería con la lista de todos los sources y headers files que se usaran para crear la librería.
* **ADD\_LIBRARY:** Crea una librería estática, que no es más que un archivo con extensión .a llamado optionparserlib usando como sources a la variable librería.
* **TARGET\_INCLUDE\_DIRECTORIES:** Especifica que esta librería puede ser usado de forma pública por cualquier source file.

Es importante aclarar que al momento de importar la biblioteca, la instrucción **#include** deberá hacerse usando angle brackets (< >) y no comillas dobles (“ ”).



# GOOGLE TESTS

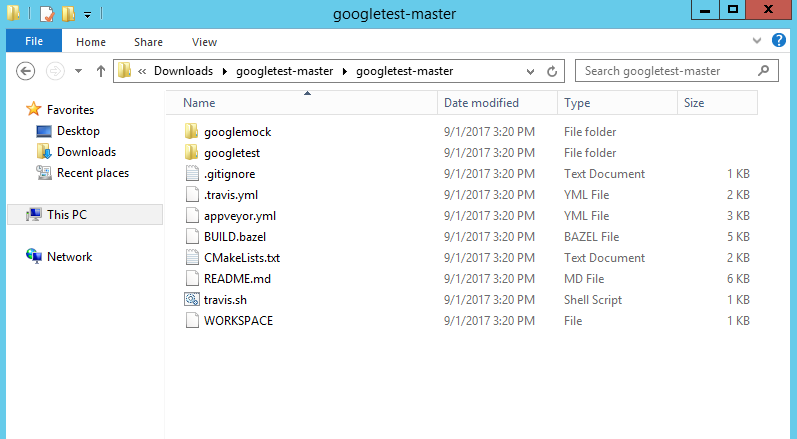
# Instalación de Google Tests

Ingresa al sitio web, <https://github.com/google/googletest> y clona o descarga la carpeta del google test.

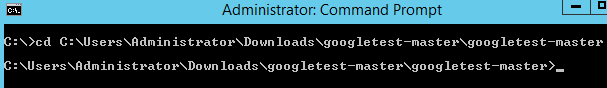
Una vez clonado o descargado el google test nos dirigimos a su directorio hasta estar a su raíz a través del cmd.

Por ejemplo en el caso de descarga lo más probable es que este en la siguiente dirección:

C:\Users\Administrator\Downloads\googletest-master\googletest-master

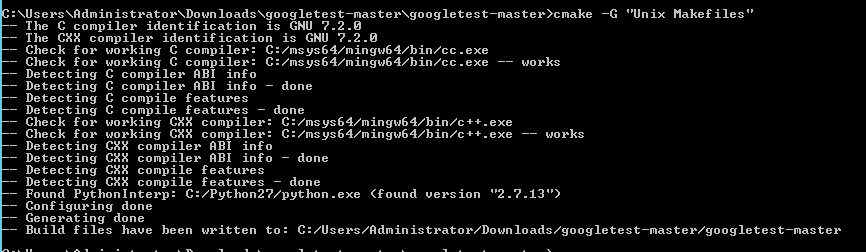


Posteriormente abrimos nuestro cmd y nos dirigimos a esa misma dirección.

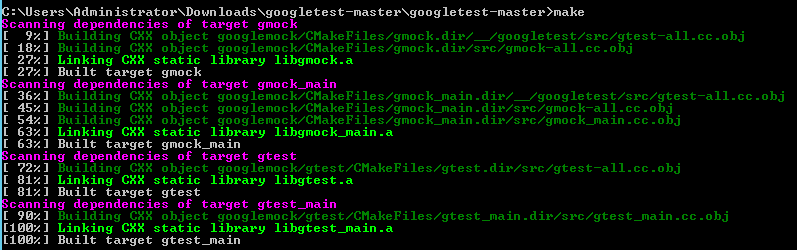


Ejecutamos nuestro cmake con la siguiente sentencia.

Cmake -G "Unix Makefiles"



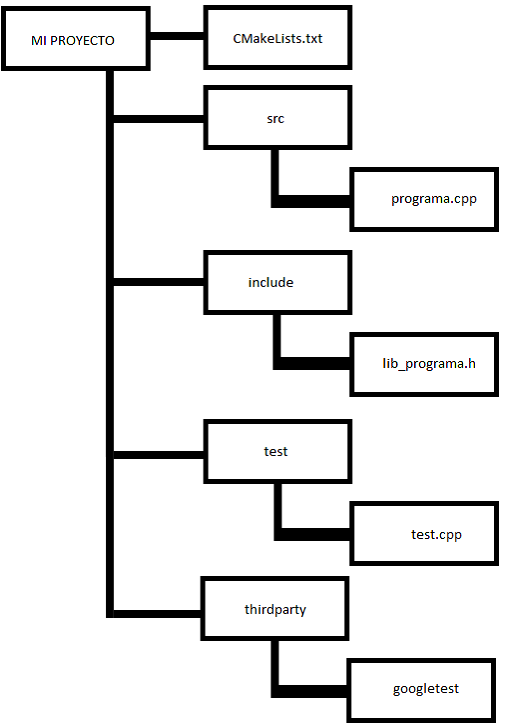
De esta manera se ejecutara el archivo CMakeLists.txt que tiene ahí dentro en el google test, posteriormente ejecutamos la sentencia make.



Con eso finaliza la instalación del google test, es importante que para poder usar el google test se tiene que realizar este procedimiento.

# Unit Tests con Google Tests

Para realizar unit testing con google tests usando cmake, vamos a realizar un directorio de esta manera: (EL GOOGLE TEST YA DEBERIA ESTAR CON SU BUILDING).



El cual el directorio de mi proyecto tendrá un txt (CMakeLists.txt) para realizar el build, una carpeta con el nombre de src donde contendrá nuestros programas principales, una carpeta con el nombre de include donde contendrá nuestras cabeceras de nuestros programas, una carpeta test el cual contendrá nuestro programa para testear las funciones de nuestros cabeceras, y finalmente una carpeta thirdparty donde tendrá la instalación del googletest.

Para nuestro ejemplo, vamos a realizar una función llamada cubic que el objetivo de esa función es retornar el cubico de un número, por ejemplo: el cubico de 2 es 8, (2 = 2 \* 2 \* 2).

Empezaremos a construir nuestros programas.

Para nuestro programa principal (programa.cpp) pondremos este código:

#include "lib\_programa.h"

#include <cmath>

double cubic(double d)

{

return pow (d,3);

}

Para nuestro encabezado (lib\_programa.h) pondremos este código:

#pragma once

double cubic (double);

Para nuestro programa para el test (test.cpp) pondremos este código:

#include "gtest/gtest.h"

#include "lib\_programa.h"

TEST(Cubic3, myCubeTest)

{

ASSERT\_EQ(25, cubic(3));

}

TEST(Cubic10, myCubeTest)

{

ASSERT\_EQ(1000, cubic(10));

}

int main(int argc, char \*\* argv)

{

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

Con eso ya tenemos nuestros programas para el funcionamiento y ahora aplicaremos el google test para verificar que nuestra función cubic efectivamente retorne el cubico de un número, en nuestro test.cpp el test Cubic3 es un test que tiene que fallar porque el cubico de 3 es 27 no 25, y el test Cubic10 es un test que tiene que pasar porque efectivamente el cubico de 10 es 1000.

Para ello nuestro CMakeLists.txt pondremos este código:

CMAKE\_MINIMUM\_REQUIRED(VERSION 3.0.0)

PROJECT(basic\_test)

INCLUDE\_DIRECTORIES(include)

ADD\_SUBDIRECTORY(thirdparty/googletest)

ENABLE\_TESTING()

INCLUDE\_DIRECTORIES(${gtest\_SOURCE\_DIR}/include ${gtest\_SOURCE\_DIR})

SET(src\_program test/test.cpp src/programa.cpp)

ADD\_EXECUTABLE(basic\_test ${src\_program})

TARGET\_LINK\_LIBRARIES(basic\_test gtest gtest\_main -lpthread)

ADD\_TEST(basic\_test testi.exe)

En el cual los funcionamientos para el CMakeLists.txt, muchos de ellos ya fueron explicados atrás, sin embargo, existen nuevas funciones como:

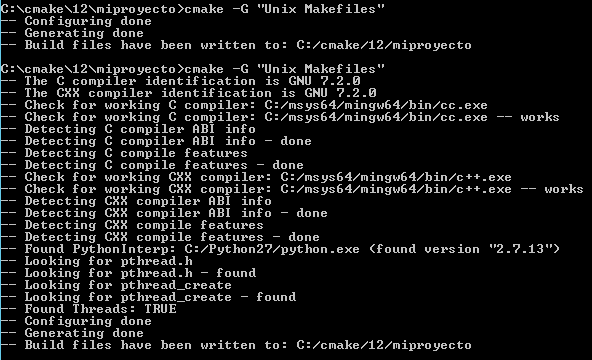
ENABLE\_TESTING() -> Sirve para habilitar el test en ese directorio.

ADD\_TEST(<name> <config>) -> Sirve para dar nombre a tu test, y crear el ejecutable, el nombre no puede tener espacios, ni caracteres especiales.

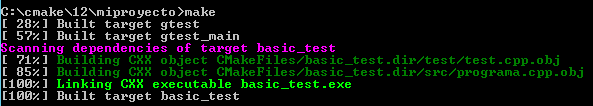
Entramos a nuestro cmd, nos ubicamos en la dirección del proyecto.



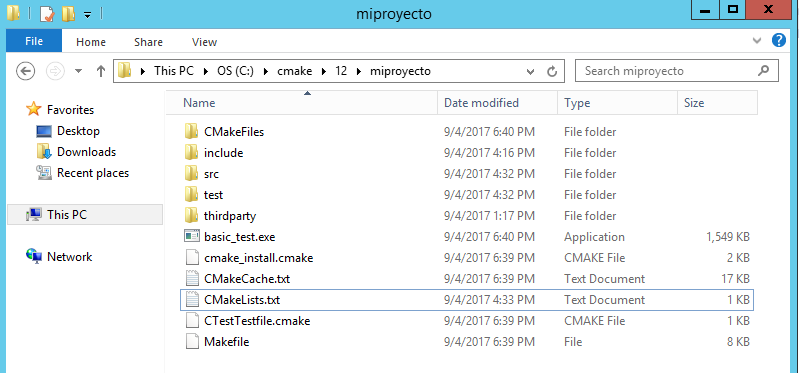
Escribimos cmake -G "Unix Makefiles" en el cmd para que el cmake realice el procedimiento con el CMakeLists.txt.



Posteriormente realizamos el make y con eso se creara el building.



Si vemos en nuestro directorio de nuestro proyecto, se podra observar que se creo un archivo basic\_text.exe.



Y si lo ejecutamos en el cmd, nos debera mostrar los test que pasaron y los test que fallaron, en nuestro caso sera uno fallido y uno que pase.

