Compilando C/C++ con CMake

Jeffrey Alexander Argueta Galicia
Christopher Briyant Álvarez Moz
Facultad de Ingeniería y Ciencias Naturales, Ingeniería en Sistemas

Miércoles 9 de noviembre de 2024

Índice

CMake: El Sistema de Compilación Estándar

> *Lo que pone a CMake por encima de sus competidores: su extensa lista de características.*
>
> **— *Bryce Adelstein Lelbach***

Definición

CMake es una herramienta de construcción de código abierto y multiplataforma, creada para simplificar y optimizar la gestión de proyectos de software, especialmente aquellos desarrollados en lenguajes como C y C++. Su diseño busca proporcionar una interfaz coherente y portátil que permita a los desarrolladores definir cómo se debe construir su software, independientemente del sistema operativo o del entorno de desarrollo que estén utilizando.

¿Por qué CMake?

Si alguna vez has mantenido el proceso de compilado e instalación de un paquete de software, entonces estarás interesado en CMake. A diferencia de las herramientas de construcción tradicionales, que a menudo requieren scripts específicos para cada plataforma (como Make, Ant o NMake), CMake adopta un enfoque declarativo. Esto significa que los desarrolladores pueden especificar qué quieren construir y cómo, sin preocuparse por los detalles técnicos de la plataforma subyacente. Esta portabilidad es uno de los puntos más destacados de CMake, ya que permite a un mismo proyecto ser compilado en múltiples entornos con un solo archivo de configuración.

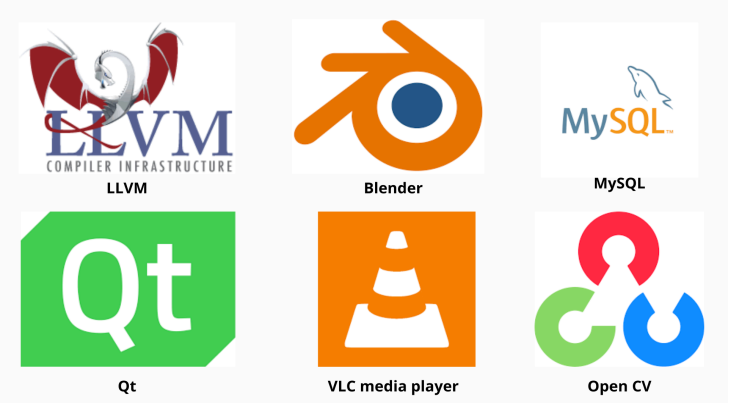
Historia de CMake

En el año 2000 CMake en respuesta a la necesidad de un entorno de compilación multiplataforma potente para The Insight Toolkit (ITK) y Visualization Toolkit (VTK). Ha evolucionado en las últimas dos décadas con muchas mejoras y nuevas características. Sigue siendo fuertemente apoyada por la comunidad de usuarios, incluyendo al autor original y líder de arquitectura, Bill Hoffman, y expertos internos de Kitware.

2000	CMake se crea para el Insight Toolkit Project de la National Library of Medicine's (NLM)
2006	KDE se cambia a CMake
2008	Primer CMakeLists.txt para el compilador LLVM
2016	LLVM cambia a CMake, eliminando autoconf

2019	El Qt group se cambia a CMake
2022	CMake es nombrado el sistema de construcción estándar por Bryce Adelstein Lelbach durante su charla de conferencia de C++Now - "¿Qué pertenece en la librería estándar de C++?"

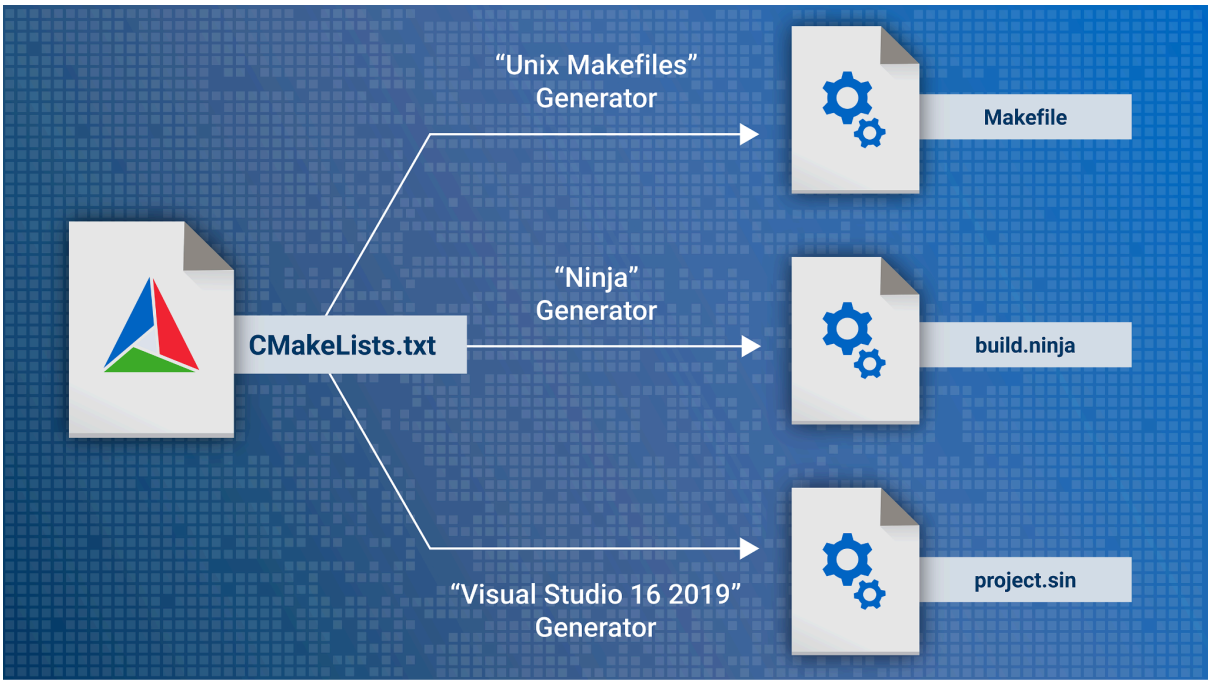
Aplicaciones importantes que utilizan CMake



Características

- ## Un solo archivo se construye en múltiples plataformas

CMake permite a los desarrolladores describir cómo construir sistemas de software simples y muy complicados con un solo conjunto de archivos de entrada. Esto se puede utilizar para construir el software en múltiples plataformas, desde Android a iOS a sistemas de computación de alto rendimiento.



- ## Enfoque centrado en el objetivo

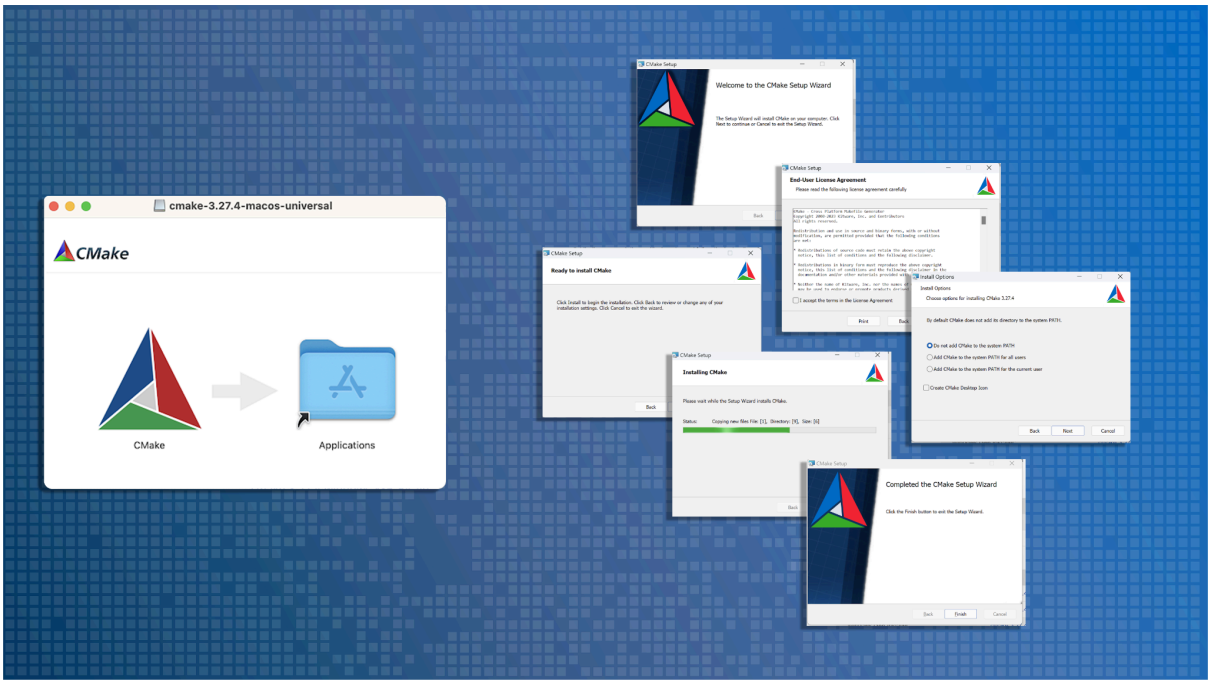
CMake permite especificar la construcción como un conjunto de objetivos (ejecutables, bibliotecas, comandos personalizados). Cada objetivo define de qué otros objetivos depende directamente. CMake entonces realiza la construcción en el orden correcto e incluye objetivos de enlace transitivos.

- ## Sistemas de construcción de múltiples objetivos

CMake admite múltiples sistemas de compilación de objetivos, incluyendo Visual Studio, Xcode, ninja, make, y VSCode. Permite a un proyecto utilizar su recurso más importante, el desarrollador, a todo su potencial. Como CMake admite muchos sistemas IDE populares para las herramientas de construcción de C++, así como herramientas de compilación de línea de comando, los desarrolladores son capaces de elegir la herramienta de compilación con la que son más productivos.

- ## Sistema de empaquetado multiplataforma

CMake contiene el sistema CPack, que permite la creación de instaladores multiplataforma para Linux, Windows y Mac.



- ## Sistema de instalación multiplataforma completa

CMake contiene un sistema de instalación multiplataforma. Con el mismo conjunto de comandos, se puede instalar un paquete de software en Linux, Windows y Mac.

Instalación de CMake

Antes de usar CMake necesitas instalarlo o compilar el binario de CMake en tu sistema. En muchos sistemas encontrarás que CMake ya está instalado, o está disponible para instalar con la herramienta de manejo de paquetes estándar para el sistema. Cygwin, Debian, FreeBSD, Mac OS X Fink, y muchos otros poseen distribuciones de CMake. Si tu sistema no tiene un paquete de CMake, puedes encontrar un precompilado CMake para la mayoría de arquitecturas en https://cmake.org/. Si no encuentras el binario precompilado para tu sistema, entonces puedes compilarlo. Para compilar CMake necesitarás un compilador moderno de C++.

Instalación binaria en UNIX y Mac

Si tu sistema provee CMake como uno de sus paquetes estándar, entonces sigue las instrucciones de instalación de tu sistema de paquetes. Si tu sistema no tiene CMake, o tiene una versión deprecada de CMake, puedes descargar el binario precompilado desde https://cmake.org/. Los binarios vienen de forma comprimida como archivo tar. El archivo tar contiene un README y un archivo tar encerrado. El archivo README contiene un manifiesto de los archivos contenidos en el archivo tar, y algunas instrucciones. Para instalar, simplemente extrae el archivo tar encerrado en su directorio destino (normalmente `/usr/local`). Sin embargo, puede ser cualquier directorio, y no requiere privilegios root para la instalación.

Instalación binaria en Windows

Para Windows CMake tiene un archivo NullSoft install para descargar de https://cmake.org/. Para instalar este archivo, simplemente ejecuta en tu máquina de Windows en la cual quieres CMake. Serás capaz de iniciar CMake desde el Menú de Inicio después de la instalación.

Compilando CMake por ti mismo

Si los binarios no están disponibles para tu sistema, o la versión de CMake que deseas usar no está disponible, puedes compilar CMake desde su código fuente. Puedes obtenerlo siguiendo las instrucciones en https://cmake.org/. Puedes compilarlo de dos maneras diferentes. Si tienes una versión en tu sistema de CMake, puedes usarla para compilar otras versiones de CMake. El desarrollo actual de CMake puede siempre ser compilado de anteriores versiones. La segunda manera de compilar CMake es corriendo su bootstrap build script. Para hacer esto cambia tu directorio hacia tu directorio fuente de CMake y escribe

```
./bootstrap
make
```

```
make install
```

Si no estás ocupando el compilador GNU C++, tendrás que comentarle al bootstrap script cuál compilador usar. Esto se hace configurando la variable de entorno `CXX` antes de correr bootstrap. Si necesitas usar alguna bandera específica con tu compilador, establece la variable de entorno `CXXFLAGS`. Por ejemplo, en el SGI con el compilador 7.3X, compilarías CMake tal que así:

```
cd CMake
(setenv CXX CC; setenv CXXFLAGS "-LANG:std"; ./bootstrap)
make
make install
```

Uso y Sintaxis básica de CMake

Usar CMake es simple. El proceso de compilado es controlado creando uno o más CMakeLists files (realmente es CMakeLists.txt pero esta guía dejará afuera la extensión en la mayoría de casos) en cada uno de los directorios que crean un proyecto. Los archivos CMakeLists deben contener la descripción del proyecto en el lenguaje simple de CMake. El lenguaje está expresado como una serie de comandos. Cada comando es evaluado en el orden que aparece en el archivo CMakeLists. Los comando poseen la forma

```
command (args…)
```

Dónde `command` es el nombre del comando, y `args` es la lista de argumentos separados por espacios blancos. (Argumentos con espacio en blanco incluído deben ser citados con comillas dobles.) CMake es de tipado insensible para los nombres de los comandos a partir de la versión 2.2. Dónde ves `command` puedes usar `COMMAND` ó `Command`. Versiones viejas de CMake solo aceptan comandos en mayúsculas.

Hello World con CMake

Para comenzar consideremos el archivo CMakeLists más simple posible. Para compilar un ejecutable de un archivo fuente el CMakeLists debe contener 3 líneas:

```
cmake_minimum_required (VERSION 3.10)
project (HELLO)
add_executable (Hello Hello.cpp)
```

El comando `cmake_minimum_required` indica que la persona corriendo CMake en tu proyecto deberá tener mínimo la versión 3.10. `project` indica que el

nombre del espacio de trabajo resultante debe ser y `add_executable` añade un objetivo de ejecutable al proceso de construcción.

Variables simples

Variables son referenciadas usando `${VAR}` sintaxis. Múltiples argumentos pueden ser agrupados usando el comando `set`. Todos los comandos expanden las listas, por ejemplo, `set(Foo a b c)` va a resultar en configurando la variable Foo para `a b c`, y si `Foo` es pasado a otro comando `command(${Foo})` eso es equivalente a `command(a b c)`. Si quieres pasar una lista de argumentos a un comando como si fuera un solo simple argumento, citado entre comillas. Por ejemplo `command("${Foo}")` será invocado pasando solamente el argumento equivalente a `command("a b c")`.

Construyendo tu proyecto

```
cmake <tu directorio CMakeLists>
cmake --build <tu directorio del binario>
```

Eso es todo lo que hay para instalar y correr CMake para proyectos simples.

Objetivos - Targets

Ahora que ya hemos discutido el proceso general de CMake, consideremos uno de los items claves almacenados en la instancia `cmMakefile`. Probablemente el ítem más importante son los objetivos. Los objetivos representan ejecutables, librerías, y utilidades construidas en CMake. Cada `add_librery`, `add_executable`, y `add_custom_target` comando crea un objetivo. Por ejemplo, el siguiente comando creará un objetivo llamado foo que es una librería estática, con `foo1.cpp` y `foo2.cpp` como archivos fuente.

```
add_library (foo STATIC foo1.cpp foo2.cpp)
```

Objetivos almacenan una lista de librerías que enlazan las cuales son establecidas usando el comando `target_link_libraries`. Nombres pasados a este comando pueden ser librerías, paths completos a librerías, o el nombre de una librería del comando `add_library`. También almacenan directorios enlace para usar cuando se enlaza, la ubicación de instalación del objetivo, y comandos personalizados a ejecutar después de enlazar.

```
add_library (foo foo.cxx)
target_link_libraries (foo bar)

add_executable (foobar foobar.cxx)
target_link_libraries (foobar foo)
```

Entornos de desarrollo con CMake

CMakeLists

Ejemplo 1.

<Directorio del proyecto>

```
/<Proyecto>
├─ build
│  └─ <Proceso de compilado>
├─ main.cpp
└─ CMakeLists.txt
```

main.cpp

```
#include <iostream>

int main (int argc, char *argv[]) {
     std::cout << "Hello World!" << std::endl;
     return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(App)

add_executable(App main.cpp)
```

Ejemplo 2.

<Directorio del proyecto>

```
/<Proyecto>
├─ build
│   └─ <Proceso de compilado>
├─ include
│   └─ hello.hpp
├─ src
│   └─ main.cpp
└─ CMakeLists.txt
```

hello.hpp

```
#ifndef HELLO_H
#define HELLO_H

#include <iostream>

void hello() {
     std::cout << "Hello World!" << std::endl;
}

#endif // !HELLO_H
```

main.cpp

```
#include "../include/hello.hpp"

int main (int argc, char *argv[]) {
     hello();
     return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(App)

set(SRC_DIR ${PROJECT_SOURCE_DIR}/src)
set(SRC_LST ${SRC_DIR}/main.cpp)

add_executable(${PROJECT_NAME} ${SRC_LST})

target_include_directories(${PROJECT_NAME} PUBLIC
${PROJECT_SOURCE_DIR}/include)
```

Ejemplo 3.

<Directorio del proyecto>

```
/<Proyecto>
├─ build
│   └─ <Proceso de compilado>
├─ include
│   └─ hello.hpp
├─ src
│   └─ main.cpp
├─ build.bat
└─ CMakeLists.txt
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

project(App LANGUAGES CXX)

set(SRC_DIR ${PROJECT_SOURCE_DIR}/src)
set(SRC_LST ${SRC_DIR}/main.cpp)
set(INC_DIR ${PROJECT_SOURCE_DIR}/include)

add_executable(${PROJECT_NAME} ${SRC_LST})

target_include_directories(${PROJECT_NAME} PUBLIC
${INCLUDE_DIR})
```

build.bat

```
mkdir build
cd build
cmake ..
cmake --build .
cd debug
App.exe
```

Proyecto wxWidget con CMakeLists

Ejemplo 1.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(ExamenFinalV2 VERSION 1.0)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

include_directories(${PROJECT_SOURCE_DIR}/include)
include_directories(${PROJECT_SOURCE_DIR}/include/class)
include_directories(${PROJECT_SOURCE_DIR}/include/clientes)
include_directories(${PROJECT_SOURCE_DIR}/include/pagos)
include_directories(${PROJECT_SOURCE_DIR}/include/prestamos)

find_package(wxWidgets REQUIRED COMPONENTS core base)

include(${wxWidgets_USE_FILE})

file(GLOB_RECURSE SRC_FILES
     ${PROJECT_SOURCE_DIR}/src/*.cpp
)

add_executable(${PROJECT_NAME} ${SRC_FILES})

target_link_libraries(${PROJECT_NAME} ${wxWidgets_LIBRARIES})
```

Ejemplo 2.

src/CMakeLists.txt

```
add_executable(MainApp MainApp.cpp)

target_include_directories(
     MainApp
     PRIVATE ${CMAKE_SOURCE_DIR}/ui
)

target_link_libraries(
     MainApp
     PRIVATE UiLib UtilsLib ${wxWidgets_LIBRARIES}
)
```

ui/CMakeLists.txt

```
add_library(UiLib MainFrame.cpp MainFrame.h)

target_include_directories(
     UiLib
     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}
)
```

utils/CMakeLists.txt

```
add_library(UtilsLib Helper.cpp Helper.h)

target_include_directories(
     UtilsLib
     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}
)
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(EjemploUno)

find_package(wxWidgets REQUIRED COMPONENTS core base)

include(${wxWidgets_USE_FILE})

add_subdirectory(src)
add_subdirectory(ui)
add_subdirectory(utils)
```

Ejemplo 3.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(EjemploCPack VERSION 1.0)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

include_directories(${PROJECT_SOURCE_DIR}/include)
include_directories(${PROJECT_SOURCE_DIR}/include/class)
include_directories(${PROJECT_SOURCE_DIR}/include/clientes)
include_directories(${PROJECT_SOURCE_DIR}/include/pagos)
include_directories(${PROJECT_SOURCE_DIR}/include/prestamos)

find_package(wxWidgets REQUIRED COMPONENTS core base)

include(${wxWidgets_USE_FILE})

file(GLOB_RECURSE SRC_FILES
     ${PROJECT_SOURCE_DIR}/src/*.cpp
)

add_executable(${PROJECT_NAME} ${SRC_FILES})

target_link_libraries(${PROJECT_NAME} ${wxWidgets_LIBRARIES})

install(TARGETS ${PROJECT_NAME} DESTINATION /usr/bin)

set(CPACK_PACKAGE_NAME "EjemploCPack")
set(CPACK_PACKAGE_VERSION "1.0.0")
set(CPACK_PACKAGE_DESCRIPTION "Un ejemplo de uso de CPack")
set(CPACK_PACKAGE_CONTACT "tesst@example.com")
set(CPACK_DEBIAN_PACKAGE_MAINTAINER "Christopher Álvarez")
set(CPACK_GENERATOR "DEB")

include(CPack)
```

Material impartido – Repositorio

Conclusión

En este taller, hemos explorado a fondo CMake, una herramienta esencial para la construcción y gestión de proyectos de software multiplataforma. A lo largo de la sesión, discutimos desde conceptos básicos como la definición y la sintaxis de CMakeLists.txt hasta la creación de proyectos modulares con ejemplos concretos de wxWidgets. Además, aprendimos cómo CMake simplifica el proceso de construcción en múltiples plataformas y entornos, lo que lo convierte en una opción estándar en la industria.

CMake no solo facilita la portabilidad de los proyectos, sino que también permite a los desarrolladores optimizar el flujo de trabajo mediante un sistema de construcción centrado en objetivos y altamente personalizable. Esto es especialmente útil en proyectos grandes y complejos, donde la automatización y la claridad en los procesos de compilación son claves para mantener la calidad y consistencia.

Al finalizar, comprendemos que herramientas como CMake no solo mejoran la productividad, sino que también permiten una mayor flexibilidad al trabajar con múltiples plataformas, lenguajes y entornos de desarrollo. El dominio de estas herramientas es un paso importante para cualquier ingeniero de software que busque trabajar en proyectos robustos y de gran escala.

Bibliografía

CMake Reference Documentation (s.f.). CMake.
Ken Martin & Bill Hoffman (2015). Mastering CMake (3.1 ed.). Kitware