Tutorial de CMake

Daniel Molina Cabrera

June 27, 2009

Contents

1 Objetivo del Documento

Este documento tiene como objetivo servir de tutorial a la herramienta de construcción *CMake*.

CMake es una herramienta desarrollada inicialmente para poder compilar de forma multiplataforma la librería VTK como un desarrollo interno, pero que ha sido publicada como herramienta de *Software Libre*, disponible en `http://www.cmake.org`.

El objetivo de este documento no es el de servir como una referencia completa a la documentación del sistema, ya que éste está perfectamente documentado en la página anterior. Ni servir como documentación introductoria al uso.

El objetivo es otro, es el de, partiendo de un conocimiento de la herramienta basada en la propia experiencia del autor, ofrecer una guía para su aprendizaje. Por tanto, destacaremos los aspectos que consideramos más esenciales, así como ofreceremos ejemplos propios. De esta forma, este documento puede interpretarse como un documento complementario al resto de documentación adicional existente (no sólo se explica un conjunto de las distintas opciones posibles, sino que, además, las opciones de las distintas directivas comentadas no son exhaustivas).

El motivo es que aunque la herramienta está bien documentada, la documentación se reduce a una explicación exhaustiva de las distintas directivas, sin establecer un orden de aprendizaje que permita ir avanzando poco a poco en su uso, tal y como se hace en este documento.

Nótese que en todo el documento consideraremos al lector como un desarrollador interesado en aplicar el CMake en sus propios proyectos, perfil que determina. Nota: Aunque haga referencia a Windows en el documento, advierto que aunque tengo referencias de su buen comportamiento no lo he utilizado en ese entorno (porque no tengo Windows instalado en ninguna de las máquinas que uso ;-)).

2 Configuración: CMakeLists.txt

Para aplicar la herramienta para compilar un proyecto sólo es necesario generar un fichero *CMakeLists.txt*. Dicho fichero es el encargado de indicar toda la información necesaria:

- Los ficheros a compilar.
- Las librerías y ejecutables a generar.
- Librerías de las que depende.
- Parámetros adicionales.
- Programas externos a utilizar (*doxygen*, ...).

Toda esta información se concentra en un único fichero por proyecto, y con una sintaxis que se planteó para que fuese sencilla, pero potente. Sin embargo, se puede dividir.

Es decir, si tenemos el proyecto dividido de forma jerárquica en varios directorios podemos crear un CMakeLists.txt por cada directorio, y hacer que un CMakeLists.txt llame a los CMakeLists.txt de sus subdirectorios (mediante la instrucción *ADD_SUBDIRECTORY*).

A continuación describimos los parámetros más importantes.

Notación: El uso de <> identifica un parámetro. El uso de corchetes determina que dicho parámetro es opcional

3 Configuración Mínima

Para crear un *CMakeLists.txt*, el formato mínimo suele ser:

- CMakeLists.txt mínimo:

```
PROJECT(hello)
ADD_EXECUTABLE(hello hello.c)
```

  - **PROJECT(**$<$ *proyecto* $>$ [*lenguaje*]**)** : Definición del proyecto.
    - $<$ *proyecto* $>$ es el nombre del proyecto. Asociado a dicho nombre se generarán un conjunto de variables que identifican los distintos directorios. Es un requisito necesario.
    - *lenguaje* es el lenguaje del proyecto. Aunque no es necesario puede ser muy conveniente, si se especifica CXX, por ejemplo, usa el compilador de c++ incluso aunque los ficheros sean .c. Este parámetro presupone el mismo lenguaje para todos los ficheros a compilar. Si hay ficheros en distintos formatos, no debe de indicarse.
  - **ADD_EXECUTABLE(**$<$ *ejecutable* $>$ $<$ *fichero*1 $>$ [... $<$ *ficheros* $>$]: Define un ejecutable.
    - $<$ *ejecutable* $>$, nombre del ejecutable resultante de la compilación, sin ruta. Lo crea en el mismo directorio desde el que se hay ejecutado *cmake* $<$ *arg* $>$. Sin embargo, posteriormente se puede indicar el directorio en el que instalarlo, mediante la instrucción **INSTALL**.
    - $<$ *ficheros* $>$, nombre de los ficheros de los que depende para compilar. Dado que a veces el ejecutable coincide en nombre con uno de los ficheros, dicho fichero *main* se suele indicar en esta línea indicando su extensión. Los ficheros son una lista de nombres separados por espacio en blanco, pueden tener extensión o no.Para el resto de ficheros, se suele usar una variable *SRC*, para mayor comodidad.

- **SET** (< *variable* > [< *ficheros* >]): Define variable. Este es una de las directivas más útiles para hacer el fichero legible.
  * < *variable* >, nombre de la variable. Existen varias convenciones que se recomienda seguir. Si contiene ficheros a compilar, debe terminar en *SRC*. Una vez que se haya definido, para usarla se debe de indicar $ < *variable* > $.
  * < *ficheros* > indica el conjunto de ficheros a utilizar, separados por espacios (o retornos de carro, para mayor legibilidad). También se utiliza para definir variables propias del CMake, ver Apartado 10.

  Puede usarsen variables para definir otras variables. Ejemplo:

```
SET (PROBLEM_SRC
     functions
     utils
     problem_common
)

SET (CEC_SRC
     ${PROBLEM_SRC}
     problemcec2005
)

⋮
```

- **cmake_minimum_version**(*VERSION* < *numero_version* >): Define la versión de CMake requerida.

  A menudo, el CMake exige que aparezca una línea equivalente a esta al principio del fichero *CMakeLists.txt*. El objetivo es verificar que la versión del CMake usada es igual o superior a la exigida. Esto se hace ya que el comportamiento del CMake puede variar entre versiones.

  - < *numero_version* >: Número de versión. La versión actual es la 2.6, por lo que lo normal es poner ese número. Sin embargo, la versión anterior, la 2.4, es bastante usada y son muy compatibles, por lo que puede usarse también.

4 Primer ejemplo

Haciendo uso de las directivas anteriores, vamos a crear el CMakeLists.txt de un proyecto real, pero sencillo. Copiamos el fichero cma.tgz, y crearemos un CMakeLists.txt para poder compilarlo. Consejo: Usar la directive **CMAKE_VERBOSE_MAKEFILE** del apartado 10.

5 Usando y definiendo librerías

Hasta ahora hemos utilizado CMake para construir programas en los que no había implicada ninguna librería. Vamos a ver ahora cómo tratar con librerías.

El primer paso es poder enlazar con librerías existentes. Ahora mismo no nos vamos a preocupar en buscarlas instaladas, eso se enseña en el apartado **??**.

El primer paso es indicarle que para crear el ejecutable debe de enlazar con las librerías.

- **TARGET_LINK_LIBRARIES(**$< ejecutable >$ [$< librerias >$]**)**: Indica las librerías con las que enlazar.

  - $< ejecutable >$: Nombre del ejecutable utilizado en *ADD_EXECUTABLE*.
  - $< librerias >$. En Unix/Linux por convención las librerías empiezan por libxxx.so ó libxxx.a, por lo que ni prefijo *lib* ni la extensión hay que añadirlo. Ejemplo, para compilar el programa distancia_euclidea con la librería matemática libm, se usaría *TARGET_LINK_LIBRARIES(distancia_euclidea m)*.

  Mediante esta instrucción se indica que enlace con una librería. Las librerías serán DLL (Windows), librerías dinámicas (Unix/Linux) o estáticas. Normalmente se hace uso de una variable LIBS en la que se añaden el total de librerías de las que depende.

Otro uso de librerías es el de crear vuestras propias librerías. Esta es una tarea que suele ser muy difícil con otros entornos. Con CMake es muy fácil. Simplemente se usa la directiva

- **ADD_LIBRARY(**$< nombre\_libreria > SHARED|STATIC < ficheros >$**)**: Define una librería.

  - $< nombre\_libreria >$ define el nombre de la librería (sin prefijo *lib* ni extensión).
  - $SHARED|STATIC$ define el tipo de librería. $SHARED$ define una librería dinámica (*.so* en Unix/Linux, *DLL* en Windows), mientras que $STATIC$ definiria una librería *.a*. Dada la nula dificultad en hacer la librería dinámica, recomiendo crearlas dinámicas.

```
PROJECT(hello C)
SET(LIBSRC
    hellolib)
SET(SRC
    hello)
ADD_LIBRARY(hellolib SHARED ${LIBSRC})
ADD_EXECUTABLE(hello ${SRC})
TARGET_LINK_LIBRARIES(hello hellolib)
```

6 Búsqueda de Librerías

Un aspectos que hace las herramientas *autotools* muy bien es el de buscar librerías necesarias para compilar. Antes de compilar, identifica el compilador, y las distintas librerías (ej: Qt). En el caso de no encontrarlo, termina el proceso avisando al usuario con un mensaje adecuado, de esta forma sabe que debe de instalar dicha librería antes de reintentarlo. Usando un simple Makefile daría un error en el enlazado, pero sin avisar convenientemente al usuario del motivo del error.

CMake también permite realizar este tipo de operaciones, tanto para búsqueda de ejecutables como para búsqueda de librerías. Aunque posee instrucciones para realizar búsquedas personalizadas de librerías, por comodidad permiten ficheros de extensiones que permiten buscar cómodamente una librería o programa.

Para ello se hace uso de la directiva INCLUDE conjuntamente con un fichero *Find* existente. En el siguiente apartado pongo un ejemplo.

6.1 Incluyendo generación de código mediante Doxygen

Para añadir a makefile la opción de generar la documentación en doxygen, es necesario hacer uso de la directiva **INCLUDE**.

- **INCLUDE(FindDoxygen) [REQUIRED]**: Busca el software *Doxygen*.

  Una vez indicado, define (si existe) en la variable $DOXYGEN_EXECUTABLE$ la ruta del ejecutable Doxygen. La deja en blanco si no lo encontró.

```
# Busca el Doxygen
INCLUDE(FindDoxygen)
# Comprueba si lo ha encontrado, si no no hace nada (no da error)
IF(DOXYGEN_EXECUTABLE)
# Avisa de que lo ha encontrado
MESSAGE( STATUS "Setting Doxygen Generator" )
# Define la nueva salida "doc" como resultado de ejecutar Doxygen sobre el
# directorio actual (Doxygen debe de estar perfectamente configurado)
ADD_CUSTOM_TARGET(
doc
COMMAND ${DOXYGEN_EXECUTABLE}
VERBATIM)
ENDIF(DOXYGEN_EXECUTABLE)
```

6.2 Buscando una librería

En el apartado anterior se buscaba un ejecutable, pero el caso más habitual es la búsqueda de librerías de desarrollo. Se realiza de la misma forma, mediante la directiva INCLUDE.

Ejemplo:

- **INCLUDE(FindQt4)**.

  Esta directiva permite buscar la librería Qt4 para poder enlazarla.

La acción de cada Find puede diferir, pero por convención utilizan la mayoría el mismo esquema. Una vez incluida, define una serie de variables si la encontró (donde $< PKG >$ es el nombre del paquete/librería):

- $< PKG >$**_INCLUDE_DIRS**: Define la ruta de los includes. Usualmente se ha añadido automáticamente a $INCLUDE_DIRECTORIES$, pero en algún caso puede ser necesario incluirlo.
- $< PKG >$**_LIBRARIES**: Librerías a incluir. Se deben de especificar en la instrucción **TARGET_LINK_LIBRARIES**.
- $< PKG >$**_DEFINITIONS**: Definición de la librería.

Admite el parámetro adicional REQUIRED. Si está definido y no encuentra termina, si no está definido, simplemente no hace nada, pero se puede comprobar que tras el *INCLUDE* una de estas variables esté definida. Si no es el caso, se puede escribir un mensaje de advertencia con **MESSAGE**

- **MESSAGE("Mensaje")**: Permite escribir un mensaje.

  Esta directiva permite escribir un mensaje de texto al usuario.
- **MESSAGE(FATAL_ERROR "Mensaje")**: Permite escribir un mensaje de error, y terminar.

  Esta directiva permite escribir un mensaje de texto al usuario avisando de un error, y terminar de procesar el fichero CMakeLists.txt.

6.3 Buscando una librería *a mano*

Como ya comenté, en el caso de que no exista ningún módulo para la librería que queremos usar, tenemos dos opciones.

- Definir el módulo, y contribuir a la sociedad :-).
- Hacer uso de la directiva **FIND_LIBRARY**.
- **FIND_LIBRARY(**$< variable >$ **NAMES** $< nombre\_lib >$ **PATHS** $< rutas\_busqueda >$**)**: Busca una librería.
  - $< variable >$ Nombre de la variable.
  - $< nombre\_lib >$ Lista de nombres de la librería (sin extensión y sin prefijo *lib*).
  - $< rutas\_busqueda >$, conjunto de rutas (path) en el que buscar la librería.

El proceso es muy sencillo, busca para cada una de las rutas la librería indicada un fichero con el nombre indicado, y si lo encuentra almacena en dicha variable la ruta completa de la librería. Si no la encuentra deja la variable sin definir (se puede comprobar con **IF (NOT** < *variable* >**)**.

Ejemplo:

```
FIND_LIBRARY(TCL_LIB
             NAMES tcl
             PATHS /usr/lib /usr/local/lib)

IF (TCL_LIB)
    TARGET_ADD_LIBRARY(Hello TCL_LIB)
ENDIF (TCL_LIB)
```

6.4 Ejercicio: Compilar un programa Qt

Vamos a compilar un programa Qt, para ello vamos a tener que buscar el paquete antes de compilarlo.

7 Instalando el software

Hasta ahora el make tendría la información necesaria para compilar los programas y librerías configurados, pero no tendría información de instalación. Si queremos poder usar una orden *make install* y que se instalen los programas y/o librerías, es necesario configurarlo adecuadamente, mediante las directivas siguientes.

- **INSTALL(FILES** < *nombre_libreria* > **DESTINATION lib)**: Permite instalar una librería.
  - < *nombre_libreria* >: Define el fichero de la librería instalar (con su ruta, si se encuentra en otro directorio).
- **INSTALL(FILES** < *nombre_ejecutable* > **DESTINATION bin)**: Permite instalar un ejecutable.
  - < *nombre_ejecutable* >: Define el programa a instalar (con su ruta, si se encuentra en otro directorio).

En ambos casos, se puede especificar un único ejecutable o librería o una lista de ellos (separados por espacio en blanco). Sin mayor configuración, instalará en */usr/local*/bin y */usr/local*/lib, respectivamente. Si se desea cambiar el prefijo se puede usar la variable *CMAKE_INSTALL_PREFIX*.

8 Información del make

Un problema importante del CMake es que, a diferencia de las autotools, requiere tener instalado el cmake para compilar. Aunque a primera vista pueda parecer un problema importante, ya de antemano le exigimos al usuario que compila el gcc/g++, make, todas las librerías dinámicas necesarias, ... por lo que consideramos que no es un problema tan importante.

El Makefile generado admite varias opciones. Usando *make help* se indican las distintas opciones. Las más importantes son:

- all: Permite compilar todo el sistema, tanto los ejecutables indicados con *ADD_EXECUTABLE* como las librerías *ADD_LIBRARY*.
- clean: Borra todos los ficheros generados (no elimina los ficheros CMakexxx generados por cmake).
- rebuild_cache: Compila todo, ignorando la caché.
- depend: Comprueba las dependencias, no hace nada más.

Además, por cada programa(ejecutable, librería, fichero objeto) existe una directiva para compilarlo por separado, si se desea. Además de este comportamiento por defecto, se pueden añadir opciones mediante la directiva

Sin embargo, se pueden añadir nuestras instrucciones, mediante la instrucción **ADD_CUSTOM_TARGET**.

- **ADD_CUSTOM_TARGET(**$<$ *nombre* $>$ **COMMAND** $<$ *orden* $>$ [*VERBATIM*]**)**: Permite añadir opción al make.
  - $<$ *nombre* $>$: nombre de la opción. Para ejecutar la orden se haría *make* $<$ *nombre* $>$.
  - $<$ *orden* $>$: Sentencia shell a ejecutar, a menudo es necesario usar la ruta. Por portabilidad, es conveniente hacer uso de *Find*.
  - *VERBATIM*: Parámetro adicional que activa la información.

También se pueden definir opciones propias, leídas mediante la expresión *cmake -DVAR=valor*, mediante la directiva.

- **OPTION(**$<$ *variable* $>$ $<$ *descripcion* $>$ [*valor_defecto*]**)**: Define un parámetro.
  - $<$ *variable* $>$ nombre del la variable.
  - $<$ *descripcion* $>$ Cadena con la descripción del parámetro.
  - *valor_defecto* valor por defecto.

  Ejemplo: *OPTION(INSTALL_DOC "Set to OFF to skip build/install Documentation" ON)*, permite especificar si se desea instalar la documentación (la instala por defecto).

9 Configuración de Directorios

En primer lugar, es muy usual que un directorio de código fuente a su vez se divida en otros directorios (que definan librerías, por ejemplo). En este caso, se puede definir para cada uno de esos directorios un fichero CMakeLists.txt y establecer que el directorio padre llame a los CMakeLists.txt de sus subdirectorios.

- **ADD_SUBDIRECTORY(**< *directorio* >**)**: Añade un subdirectorio.

  - < *directorio* > directorio a añadir, con la ruta relativa.

  Permite añadir un subdirectorio. Antes de procesar el directorio actual, se procesará cada uno de los subdirectorios indicado. Toda variable definida en el directorio padre mantendrá su valor para cada uno de los CMakeLists.txt de sus subdirectorios indicados. Sin embargo, es conveniente evitar depender demasiado de dichas variables, ya que generan dependencias innecesarias.

- **INCLUDE_DIRECTORIES(** < *directorios* > **)**: Permite insertar directorios para los include.

  Mediante esta variable se pueden añadir nuevas rutas para buscar los ficheros *.h*. Si se hace uso de una sentencia *#include ¡name.h¿* debería de encontrarse el fichero *name.h* o bien en una ruta del sistema (usualmente: /usr/include/, /usr/local/include, . . .) o bien en una ruta asignada a esta variable.

- **LINK_DIRECTORIES(** < *directorios* > **)**: Permite insertar directorios en donde buscar las librerías. Mediante esta variable se pueden añadir nuevas rutas para buscar las librerías.

- **AUX_SOURCE_DIRECTORIES(** < *directorio* > < *variable* > **)**: Permite añadir en la variable < *variable* > todos los ficheros almacenados en el directorio < *directorio* >.

  - < *directorio* > directorio (usualmente subdirectorio) en donde se encuentran código fuente.
  - < *variable* > variable en donde se almacenan todos los ficheros fuente del directorio indicado.

Existe una serie de variables que permite especificar las rutas para cada elemento. Aunque no suele ser necesario suele ser conveniente definirlas si el software posee una estructura compleja.

- **CMAKE_INSTALL_PREFIX**: Ruta de instalación.

- **PROJECT_BINARY_DIR**: Directorio en donde se guarda el binario (no modificar).

- **PROJECT_SOURCE_DIR**: Directorio en donde se encuentra el código fuente (no modificar).

10 Variable de Entorno de CMake

CMake usa una serie de variables de entorno que determinan su comportamiento. Todas ellas poseen un valor por defecto, pero que pueden modificarse de dos formas:

- Mediante la directiva **SET(**$< variable > < valor >$**)**, donde valor puede ser una cadena, una lista de ficheros, o un valor numérico. Todo depende de la variable utilizada.
- Mediante la línea de comando *cmake -D*$< variable >=< valor >$. Mediante este formato se puede definir sin necesidad de reescribirlo en el CMakeLists.txt. Es una opción válida para redefinir una variable más puntualmente (como **CMAKE_VERBOSE_MAKEFILE** o **CMAKE_BUILD_TYPE**), pero no para aplicarlo de forma general.

A continuación detallo las variables que considero más importantes:

- **CMAKE_VERBOSE_MAKEFILE** $OFF|ON$: Activa mensajes de información por pantalla.

  A la hora de definir, a veces pueden producirse errores como no encontrar el código de una función. Con esta línea muestra lo que hace (los ficheros que compila, los parámetros que usa). Suele ser muy recomendable usarlo.

- **CMAKE_BUILD_TYPE** *Release*|*Debug*: Selecciona el modo de depuración.

  - Release: Activa todas las opciones de optimización (sin información de depuración). **Nota:** Desactiva también las instrucciones *assert* en C/C++.
  - Debug: Activa la información del depurador (para poder usar después). No optimiza.

  Ver apartado 11 para más información.

11 Modos de Compilación

A la hora de trabajar con el compilador existen dos modos de compilación, definidos en la variable **CMAKE_BUILD_TYPE**: Depuración (*Debug*) y despliegue (*Release*).

Es recomendable trabajar en modo Debug y emplear el modo release cuando se realice la versión final, o para hacer pruebas de rendimiento (si se usan las STL de C++, hay gran diferencias de rendimiento entre un modo u otro).

Dado que supongo que usáis un sistema de repositorio para confirmar que el CMakeLists.txt de desarrollo y de implantación coincidan, el conveniente especificar el modo de compilación desde la linea de comandos (al menos en las fases iniciales d desarrollo).

A continuación muestro algunos trucos que pueden seros interesantes.

11.1 Activar las instrucciones *assert* en modo Release

A pesar de lo comentado anteriormente, es posible hacer que las instrucciones *assert* funcionen en modo Release.

```
# define macro *DEBUG macro en modo Release
STRING(TOLOWER ${CMAKE_BUILD_TYPE} CMAKE_BUILD_TYPE_TOLOWER)
if(CMAKE_BUILD_TYPE_TOLOWER MATCHES release)
    set(NDEBUG 1)
endif(CMAKE_BUILD_TYPE_TOLOWER MATCHES release)
```

11.2 Definir vuestro propio modo

En este ejemplo defino un modo Profile, en el que se activará la directiva *"-pg"* necesaria para poder usar la aplicación de profile de C++ (si no sabeis a lo que me refiero, deberíais echarle un vistado al programa *cprof*).

```
# define macro *DEBUG macro en modo Release
STRING(TOLOWER ${CMAKE_BUILD_TYPE} CMAKE_BUILD_TYPE_TOLOWER)
IF (CMAKE_BUILD_TYPE_TOLOWER MATCHES profile)
SET (GPROF_FLAGS "-pg")
SET (FLAGS "${SHARED_FLAGS} -Wall -O -DNDEBUG ${GPROF_FLAGS}")
SET (CMAKE_CXX_FLAGS "${FLAGS} ${GPROF_FLAGS}")
SET (CMAKE_EXE_LINKER_FLAGS "${GPROF_FLAGS}")
MESSAGE(STATUS "Profile")
ENDIF (CMAKE_BUILD_TYPE_TOLOWER MATCHES profile)
```

11.3 Activas todos los avisos de compilación

Existen una serie de directivas que indican al compilador que active todos los posibles mensajes de advertencia (*warnings*), que pueden utilizarse en todos los modos. Lamentablemente, no se especifica por defecto. Para activarlo se puede utilizar la directiva:

- **SET(CMAKE_CXX_FLAGS "-Wall")** para C++.
- **SET(CMAKE_C_FLAGS "-Wall")** para ansi C.

Otra opción más portable sería:

- **ADD_DEFINITIONS("-Wall")**, pero en el caso de que queramos definir directivas distintas para C y para C++ tendríamos que usar las variables anteriores).

L

12 Opciones Avanzadas

Tal y como se ha podido ver en algunos ejemplos, se pueden ejecutar las directivas en función de condiciones. Se ha intentado que sea lo suficientemente flexible para adaptarse a las distintas necesidades, pero no a costa de hacerlo demasiado complejo.

12.1 Tipos de Condiciones

A continuación presentamos los principales tipos de condiciones:

- **DEFINED(** $< var >$ **)**: Comprueba si la variable $< var >$ está definida.
- $< variable >$ **STREQUAL** $< cadena >$: Devuelve si el contenido de la variable es la cadena indicada.
- **NOT** $< CONDICION >$: Devuelve verdadero si la condición indicada es falsa.
- $< variable > < valor >$: Devuelve verdadero si la variable $< variable >$ es numérica o lógica (ON—OFF) y su valor coincide con el indicado en $< valor >$.

12.2 Directivas Condicionales

Existe directiva condicional, que permite modificar la construcción en función de alguna información (generar documentación mediante doxygen sólo si está instalado)

- **IF(** $< CONDICION >$ **)**
  ⋮
  ENDIF($< CONDICION >$ **)**

La interpretación de la cadena es evidente. Las directivas situadas dentro del IF sólo se evaluará si la condición indicada es cierta, pero en todo caso debe de ser sintácticamente correctas.

Dado que a menudo es muy molesto tener que repetir la condición dos veces (en el IF y en el ENDIF), se puede hacer uso de la condición

- **SET (CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS TRUE)**: Permite definir las sentencias *ENDIF* sin la condición.

12.3 Directivas Iterativas

Aunque el uso de instrucciones iterativas no sea realmente necesario, hay ocasiones en las que puede rsultar muy cómodo su uso.

- **FOREACH** (< *variable* > < *valores* >)
  ⋮
  ENDFOREACH(< *variable* >)

Con la instrucción **FOREACH** se puede definir aplicar un conjunto de directivas sucesivas veces, cada vez con un valor de la variable < *variable* > distinto.

Ejemplo de su uso:

```
SET(REALEALIBS
    realea
    newmat
    realpeasy
    bbob2009
    realls)

SET (LIBS m newmat ${REALEALIBS})

SET (EAs
    chc
    de
    ssga
    pso
)

FOREACH(EA ${EAs})
    ADD_EXECUTABLE(${EA} main_${EA}.cc)
    TARGET_LINK_LIBRARIES(${EA} ${LIBS} real${EA} realpcec2005 m)
    INSTALL(TARGETS ${EA} DESTINATION bin)
    INSTALL(FILES lib/libreal${EA}.so DESTINATION lib)
ENDFOREACH(EA)
```

13 Compilando un proyecto Qt

Tal y como se ha comentado, CMake es actualmente la herramienta de construcción para proyectos KDE, y, por tanto, para Qt.

El principal problema para el uso de Qt era la necesidad del sistema *moc* (una especie de preprocesador especial), que se aborda mediante el uso del módulo de Qt4.

```
qtproject/CMakeLists.txt:

project(qtproject) # the name of your project

cmake_minimum_required(VERSION 2.4.0)

find_package(Qt4 REQUIRED) # find and setup Qt4 for this project
```

```
# tell cmake to process CMakeLists.txt in that subdirectory
add_subdirectory(src)

qtproject/src/CMakeLists.txt:

# the next line sets up include and link directories and defines some variables that
# you can modify the behavior by setting some variables, e.g.
#   set(QT_USE_OPENGL TRUE)
# -> this will cause cmake to include and link against the OpenGL module
include(${QT_USE_FILE})

# the variable "qtproject_SRCS" contains all .cpp files of this project
set(qtproject_SRCS
    main.cpp
    top.cpp
)

# tell cmake to create .moc files for all files in the variable qtproject_SRCS that
# note: this assumes that you use #include "header.moc" in your files
qt4_automoc(${qtproject_SRCS})

# create an executable file named "qtproject" from the source files in the variable
add_executable(qtproject ${qtproject_SRCS})

# link the "qtproject" target against the Qt libraries. which libraries exactly, is
target_link_libraries(qtproject ${QT_LIBRARIES})
```

14 Conclusiones

Espero que esta guía dirigida les haya servido, y que se animen a utilizar el CMake :-).