Использование CMake -подготовил Жуков Дмитрий Ильич, ШАД-212

**Введение**

В данной статье рассмотрено использование системы сборки CMake, применяемой в колоссальном количестве проектов на C/C++. Строго рекомендуется прочитать первую часть руководства во избежание непонимания синтаксиса языка CMake, явным образом фигурирующего на протяжении всей статьи.

**Запуск CMake**

Ниже приведены примеры использования языка CMake, по которым Вам следует попрактиковаться. Экспериментируйте с исходным кодом, меняя существующие команды и добавляя новые. Чтобы запустить данные примеры, установите CMake с официального сайта.

**Принцип работы**

Система сборки CMake представляет из себя оболочку над другими платформенно зависимыми утилитами (например, Ninja или Make). Таким образом, в самом процессе сборки, как бы парадоксально это ни звучало, она непосредственного участия не принимает.

Система сборки CMake принимает на вход файл CMakeLists.txt с описанием правил сборки на формальном языке CMake, а затем генерирует промежуточные и нативные файлы сборки в том же каталоге, принятых на Вашей платформе.

Сгенерированные файлы будут содержать конкретные названия системных утилит, директорий и компиляторов, в то время как команды CMake орудуют лишь абстрактным понятием компилятора и не привязаны к платформенно зависимым инструментам, сильно различающихся на разных операционных системах.

**Проверка версии CMake**

Команда cmake\_minimum\_required проверяет запущенную версию CMake: если она меньше указанного минимума, то CMake завершает свою работу фатальной ошибкой. Пример, демонстрирующий типичное использование данной команды в начале любого CMake-файла:

# Задать третью минимальную версию CMake:

cmake\_minimum\_required(VERSION 3.0)

Как подметили в комментариях, команда cmake\_minimum\_required выставляет все флаги совместимости (смотреть cmake\_policy). Некоторые разработчики намеренно выставляют низкую версию CMake, а затем корректируют функционал вручную. Это позволяет одновременно поддерживать древние версии CMake и местами использовать новые возможности.

**Оформление проекта**

В начале любого CMakeLists.txt следует задать характеристики проекта командой project для лучшего оформления интегрированными средами и прочими инструментами разработки.

# Задать характеристики проекта "MyProject":

project(MyProject VERSION 1.2.3.4 LANGUAGES C CXX)

Стоит отметить, что если ключевое слово LANGUAGES опущено, то по умолчанию задаются языки C CXX. Вы также можете отключить указание любых языков путём написания ключевого слова NONE в качестве списка языков или просто оставить пустой список.

**Запуск скриптовых файлов**

Команда include заменяет строку своего вызова кодом заданного файла, действуя аналогично препроцессорной команде include языков C/C++. Этот пример запускает скриптовый файл MyCMakeScript.cmake описанной командой:

message("'TEST\_VARIABLE' is equal to [${TEST\_VARIABLE}]")

# Запустить скрипт `MyCMakeScript.cmake` на выполнение:

include(MyCMakeScript.cmake)

message("'TEST\_VARIABLE' is equal to [${TEST\_VARIABLE}]")

В данном примере, первое сообщение уведомит о том, что переменная TEST\_VARIABLE ещё не определена, однако если скрипт MyCMakeScript.cmake определит данную переменную, то второе сообщение уже будет информировать о новом значении тестовой переменной. Таким образом, скриптовый файл, включаемый командой include, не создаёт собственной области видимости, о чём упомянули в комментариях к предыдущей статье.

**Компиляция исполняемых файлов**

Команда add\_executable компилирует исполняемый файл с заданным именем из списка исходников. Важно отметить, что окончательное имя файла зависит от целевой платформы (например, <ExecutableName>.exe или просто <ExecutableName>). Типичный пример вызова данной команды:

# Скомпилировать исполняемый файл "MyExecutable" из

# исходников "ObjectHandler.c", "TimeManager.c" и "MessageGenerator.c":

add\_executable(MyExecutable ObjectHandler.c TimeManager.c MessageGenerator.c)

**Компиляция библиотек**

Команда add\_library компилирует библиотеку с указанным видом и именем из исходников. Важно отметить, что окончательное имя библиотеки зависит от целевой платформы (например, lib<LibraryName>.a или <LibraryName>.lib). Типичный пример вызова данной команды:

# Скомпилировать статическую библиотеку "MyLibrary" из

# исходников "ObjectHandler.c", "TimeManager.c" и "MessageConsumer.c":

add\_library(MyLibrary STATIC ObjectHandler.c TimeManager.c MessageConsumer.c)

• Статические библиотеки задаются ключевым словом STATIC вторым аргументом и представляют из себя архивы объектных файлов, связываемых с исполняемыми файлами и другими библиотеками во время компиляции.

• Динамические библиотеки задаются ключевым словом SHARED вторым аргументом и представляют из себя двоичные библиотеки, загружаемые операционной системой во время выполнения программы.

• Модульные библиотеки задаются ключевым словом MODULE вторым аргументом и представляют из себя двоичные библиотеки, загружаемые посредством техник выполнения самим исполняемым файлом.

• Объектные библиотеки задаются ключевым словом OBJECT вторым аргументом и представляют из себя набор объектных файлов, связываемых с исполняемыми файлами и другими библиотеками во время компиляции.

**Добавление исходников к цели**

Бывают случаи, требующие многократного добавления исходных файлов к цели. Для этого предусмотрена команда target\_sources, способная добавлять исходники к цели множество раз.

Первым аргументом команда target\_sources принимает название цели, ранее указанной с помощью команд add\_library или add\_executable, а последующие аргументы являются списком добавляемых исходных файлов.

Повторяющиеся вызовы команды target\_sources добавляют исходные файлы к цели в том порядке, в каком они были вызваны, поэтому нижние два блока кода являются функционально эквивалентными:

# Задать исполняемый файл "MyExecutable" из исходников

# "ObjectPrinter.c" и "SystemEvaluator.c":

add\_executable(MyExecutable ObjectPrinter.c SystemEvaluator.c)

# Добавить к цели "MyExecutable" исходник "MessageConsumer.c":

target\_sources(MyExecutable MessageConsumer.c)

# Добавить к цели "MyExecutable" исходник "ResultHandler.c":

target\_sources(MyExecutable ResultHandler.c)

# Задать исполняемый файл "MyExecutable" из исходников

# "ObjectPrinter.c", "SystemEvaluator.c", "MessageConsumer.c" и "ResultHandler.c":

add\_executable(MyExecutable ObjectPrinter.c SystemEvaluator.c MessageConsumer.c

ResultHandler.c)

**Генерируемые файлы**

Местоположение выходных файлов, сгенерированных командами add\_executable и add\_library, определяется только на стадии генерации, однако данное правило можно изменить несколькими переменными, определяющими конечное местоположение двоичных файлов:

• Переменные RUNTIME\_OUTPUT\_DIRECTORY и RUNTIME\_OUTPUT\_NAME определяют местоположение целей выполнения.

• Переменные LIBRARY\_OUTPUT\_DIRECTORY и LIBRARY\_OUTPUT\_NAME определяют местоположение библиотек.

• Переменные ARCHIVE\_OUTPUT\_DIRECTORY и ARCHIVE\_OUTPUT\_NAME определяют местоположение архивов.

Исполняемые файлы всегда рассматриваются целями выполнения, статические библиотеки — архивными целями, а модульные библиотеки — библиотечными целями. Для "не-DLL" платформ динамические библиотеки рассматриваются библиотечными целями, а для "DLL-платформ" — целями выполнения. Для объектных библиотек таких переменных не предусмотрено, поскольку такой вид библиотек генерируется в недрах каталога CMakeFiles.

Важно подметить, что "DLL-платформами" считаются все платформы, основанные на Windows, в том числе и Cygwin.

**Компоновка с библиотеками**

Команда target\_link\_libraries компонует библиотеку или исполняемый файл с другими предоставляемыми библиотеками. Первым аргументом данная команда принимает название цели, сгенерированной с помощью команд add\_executable или add\_library, а последующие аргументы представляют собой названия целей библиотек или полные пути к библиотекам. Пример:

# Скомпоновать исполняемый файл "MyExecutable" с

# библиотеками "JsonParser", "SocketFactory" и "BrowserInvoker":

target\_link\_libraries(MyExecutable JsonParser SocketFactory BrowserInvoker)

Стоит отметить, что модульные библиотеки не подлежат компоновке с исполняемыми файлами или другими библиотеками, так как они предназначены исключительно для загрузки техниками выполнения.

**Работа с целями**

Как упомянули в комментариях, цели в CMake тоже подвержены ручному манипулированию, однако весьма ограниченному.

Имеется возможность управления свойствами целей, предназначенных для задания процесса сборки проекта. Команда get\_target\_property присваивает предоставленной переменной значение свойства цели. Данный пример выводит значение свойства C\_STANDARD цели MyTarget на экран:

# Присвоить переменной "VALUE" значение свойства "C\_STANDARD":

get\_target\_property(VALUE MyTarget C\_STANDARD)

# Вывести значение полученного свойства на экран:

message("'C\_STANDARD' property is equal to [${VALUE}]")

Команда set\_target\_properties устанавливает указанные свойства целей заданными значениями. Данная команда принимает список целей, для которых будут установлены значения свойств, а затем ключевое слово PROPERTIES, после которого следует список вида "<название свойства> <новое значение>":

# Установить свойству 'C\_STANDARD' значение "11",

# а свойству 'C\_STANDARD\_REQUIRED' значение "ON":

set\_target\_properties(MyTarget PROPERTIES C\_STANDARD 11 C\_STANDARD\_REQUIRED ON)

Пример выше задал цели MyTarget свойства, влияющие на процесс компиляции, а именно: при компиляции цели MyTarget CMake затребует компилятора о использовании стандарта C11. Все известные именования свойств целей перечисляются на этой странице.

Также имеется возможность проверки ранее определённых целей с помощью конструкции if(TARGET <TargetName>):

# Выведет "The target was defined!" если цель "MyTarget" уже определена,

# а иначе выведет "The target was not defined!":

if(TARGET MyTarget)

message("The target was defined!")

else()

message("The target was not defined!")

endif()

**Добавление подпроектов**

Команда add\_subdirectory побуждает CMake к незамедлительной обработке указанного файла подпроекта. Пример ниже демонстрирует применение описанного механизма:

# Добавить каталог "subLibrary" в сборку основного проекта,

# а генерируемые файлы расположить в каталоге "subLibrary/build":

add\_subdirectory(subLibrary subLibrary/build)

В данном примере первым аргументом команды add\_subdirectory выступает подпроект subLibrary, а второй аргумент необязателен и информирует CMake о папке, предназначенной для генерируемых файлов включаемого подпроекта (например, CMakeCache.txt и cmake\_install.cmake).

Стоит отметить, что все переменные из родительской области видимости унаследуются добавленным каталогом, а все переменные, определённые и переопределённые в данном каталоге, будут видимы лишь ему (если ключевое слово PARENT\_SCOPE не было определено аргументом команды set). Данную особенность упомянули в комментариях к предыдущей статье.

**Поиск пакетов**

Команда find\_package находит и загружает настройки внешнего проекта. В большинстве случаев она применяется для последующей линковки внешних библиотек, таких как Boost и GSL. Данный пример вызывает описанную команду для поиска библиотеки GSL и последующей линковки:

# Загрузить настройки пакета библиотеки "GSL":

find\_package(GSL 2.5 REQUIRED)

# Скомпоновать исполняемый файл с библиотекой "GSL":

target\_link\_libraries(MyExecutable GSL::gsl)

# Уведомить компилятор о каталоге заголовков "GSL":

target\_include\_directories(MyExecutable ${GSL\_INCLUDE\_DIRS})

В приведённом выше примере команда find\_package первым аргументом принимает наименование пакета, а затем требуемую версию. Опция REQUIRED требует печати фатальной ошибки и завершении работы CMake, если требуемый пакет не найден. Противоположность — это опция QUIET, требующая CMake продолжать свою работу, даже если пакет не был найден.

Далее исполняемый файл MyExecutable линкуется с библиотекой GSL командой target\_link\_libraries с помощью переменной GSL::gsl, инкапсулирующей расположение уже скомпилированной GSL.

В конце вызывается команда target\_include\_directories, информирующая компилятора о расположении заголовочных файлов библиотеки GSL. Обратите внимание на то, что используется переменная GSL\_INCLUDE\_DIRS, хранящая местоположение описанных мною заголовков (это пример импортированных настроек пакета).

Вам, вероятно, захочеться проверить результат поиска пакета, если Вы указали опцию QUIET. Это можно сделать путём проверки переменной <PackageName>\_FOUND, автоматически определяемой после завершения команды find\_package. Например, в случае успешного импортирования настроек GSL в Ваш проект, переменная GSL\_FOUND обратится в истину.

В общем случае команда find\_package имеет две разновидности запуска: модульную и конфигурационную. Пример выше применял модульную форму. Это означает, что во время вызова команды CMake ищет скриптовый файл вида Find<PackageName>.cmake в директории CMAKE\_MODULE\_PATH, а затем запускает его и импортирует все необходимые настройки (в данном случае CMake запустила стандартный файл FindGSL.cmake).

**Способы включения заголовков**

Информировать компилятора о располжении включаемых заголовков можно посредством двух команд: include\_directories и target\_include\_directories. Вы решаете, какую из них использовать, однако стоит учесть некоторые различия между ними (идея предложена в комментариях).

Команда include\_directories влияет на область каталога. Это означает, что все директории заголовков, указанные данной командой, будут применяться для всех целей текущего CMakeLists.txt, а также для обрабатываемых подпроектов (смотреть add\_subdirectory).

Команда target\_include\_directories влияет лишь на указанную первым аргументом цель, а на другие цели никакого воздействия не оказывается. Пример ниже демонстрирует разницу между этими двумя командами:

add\_executable(RequestGenerator RequestGenerator.c)

add\_executable(ResponseGenerator ResponseGenerator.c)

# Применяется лишь для цели "RequestGenerator":

target\_include\_directories(RequestGenerator headers/specific)

# Применяется для целей "RequestGenerator" и "ResponseGenerator":

include\_directories(headers)

В комментариях упомянуто, что в современных проектах применение команд include\_directories и link\_libraries является нежелательным. Альтернатива — это команды target\_include\_directories target\_link\_libraries, действующие лишь на конкретные цели, а не на всю текущую область видимости.

**Установка проекта**

Команда install генерирует установочные правила для Вашего проекта. Данная команда способна работать с целями, файлами, папками и многим другим. Сперва рассмотрим установку целей.

Для установки целей необходимо первым аргументом описанной функции передать ключевое слово TARGETS, за которым должен следовать список устанавливаемых целей, а затем ключевое слово DESTINATION с расположением каталога, в который установятся указанные цели. Данный пример демонстрирует типичную установку целей:

# Установить цели "TimePrinter" и "DataScanner" в директорию "bin":

install(TARGETS TimePrinter DataScanner DESTINATION bin)

Процесс описания установки файлов аналогичен, за тем исключением, что вместо ключевого слова TARGETS следует указать FILES. Пример, демонстрирующий установку файлов:

# Установить файлы "DataCache.txt" и "MessageLog.txt" в директорию "~/":

install(FILES DataCache.txt MessageLog.txt DESTINATION ~/)

Процесс описания установки папок аналогичен, за тем исключением, что вместо ключевого слова FILES следует указать DIRECTORY. Важно подметить, что при установке будет копироваться всё содержимое папки, а не только её название. Пример установки папок выглядит следующим образом:

# Установить каталоги "MessageCollection" и "CoreFiles" в директорию "~/":

install(DIRECTORY MessageCollection CoreFiles DESTINATION ~/)

После завершения обработки CMake всех Ваших файлов Вы можете выполнить установку всех описанных объектов командой sudo checkinstall (если CMake генерирует Makefile), или же выполнить данное действие интегрированной средой разработки, поддерживающей CMake.

**Наглядный пример проекта**

Данное руководство было бы неполным без демонстрации реального примера использования системы сборки CMake. Рассмотрим схему простого проекта, использующего CMake в качестве единственной системы сборки:

+ MyProject

- CMakeLists.txt

- Defines.h

- StartProgram.c

+ core

- CMakeLists.txt

- Core.h

- ProcessInvoker.c

- SystemManager.c

Главный файл сборки CMakeLists.txt описывает компиляцию всей программы: сперва происходит вызов команды add\_executable, компилирующей исполняемый файл, затем вызывается команда add\_subdirectory, побуждающая обработку подпроекта, и наконец, исполняемый файл линкуется с собранной библиотекой:

# Задать минимальную версию CMake:

cmake\_minimum\_required(VERSION 3.0)

# Указать характеристики проекта:

project(MyProgram VERSION 1.0.0 LANGUAGES C)

# Добавить в сборку исполняемый файл "MyProgram":

add\_executable(MyProgram StartProgram.c)

# Требовать обработку файла "core/CMakeFiles.txt":

add\_subdirectory(core)

# Скомпоновать исполняемый файл "MyProgram" со

# скомпилированной статической библиотекой "MyProgramCore":

target\_link\_libraries(MyProgram MyProgramCore)

# Установить исполняемый файл "MyProgram" в директорию "bin":

install(TARGETS MyProgram DESTINATION bin)

Файл core/CMakeLists.txt вызывается главным файлом сборки и компилирует статическую библиотеку MyProgramCore, предназначенную для линковки с исполняемым файлом:

# Задать минимальную версию CMake:

cmake\_minimum\_required(VERSION 3.0)

# Добавить в сборку статическую библиотеку "MyProgramCore":

add\_library(MyProgramCore STATIC ProcessInvoker.c SystemManager.c)

После череды команд cmake . && make && sudo checkinstall работа системы сборки CMake завершается успешно. Первая команда запускает обработку файла CMakeLists.txt в корневом каталоге проекта, вторая команда окончательно компилирует необходимые двоичные файлы, а третья команда устанавливает скомпонованный исполняемый файл